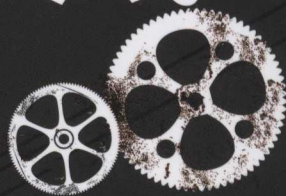


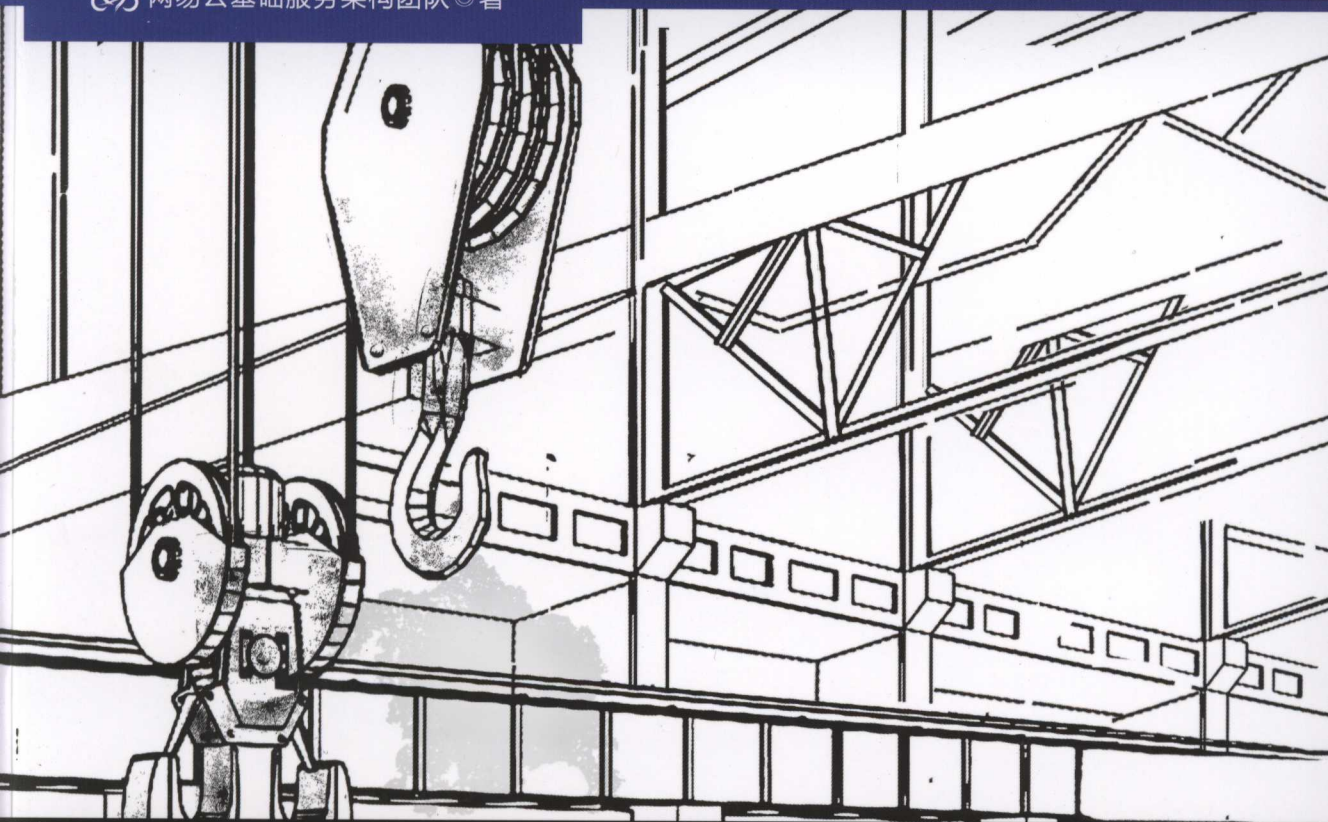
版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

云原生应用架构实践

从单体到服务化架构演进



 网易云基础服务架构团队◎著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

网易云，网易集团旗下云计算和大数据品牌，业界领先的高品质场景化云服务 and 大数据服务提供商。网易云致力于为客户提供云计算基础服务（网易蜂巢）、通信与视频（网易云信和视频云）、云安全（网易易盾）、全智能云客服（网易七鱼）等一系列场景化云服务，以及一站式大数据管理与应用开发平台（网易猛犸）和企业级大数据可视化分析平台（网易有数）等一系列大数据产品，以求更好、更快、更人性化地满足客户业务需求，帮助解决业务痛点，实现业务目标，助力企业客户成功，共创云上精彩世界。

官方网站: www.163yun.com



云原生应用架构实践

从单体到服务化架构演进



网易云基础服务架构团队◎著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

针对企业如何采用云原生架构实现高效的产品迭代能力、支持互联网业务健康发展,本书总结了一套可行的方法论。书中详解了云原生应用的内涵和要点,对实现云原生应用面临的功能和非功能(高性能、高可用、可扩展、安全性、高可靠等)的不同阶段需求和实现方案进行了较为完整的梳理。内容涵盖了系统工程化、高性能数据库、分布式数据库、DevOps、微服务架构、服务化测试、多机房架构等方面,既有业务挑战分析,也有架构实践指导,并通过实战案例加以诠释。

本书适合希望采用云计算帮助企业实现业务提升的 CTO、CIO、架构师等群体。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

云原生应用架构实践 / 网易云基础服务架构团队著. —北京: 电子工业出版社, 2017.7
ISBN 978-7-121-31516-9

I. ①云… II. ①网… III. ①云计算 IV. ①TP393.027

中国版本图书馆 CIP 数据核字(2017)第 105158 号

策划编辑: 符隆美

责任编辑: 徐津平

印 刷: 北京天宇星印刷厂

装 订: 北京天宇星印刷厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×980 1/16 印张: 23 字数: 450 千字

版 次: 2017 年 7 月第 1 版

印 次: 2017 年 7 月第 1 次印刷

印 数: 3000 册 定价: 79.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: (010) 51260888-819, faq@phei.com.cn。

前言

从互联网、移动互联网到云计算、大数据、人工智能，再到虚拟现实/增强现实，十多年来，信息技术的日新月异催生了不断涌现的互联网新业态，也推动了传统行业投身于数字化创新的浪潮。此时此刻，机器取代人类的说法固然夸张，却也真实地反映了新技术应用对产业带来的冲击。

全球顶级商业机构都是能够把握技术创新成果的企业；而对技术运用不当的企业，即便起点与前者一致，或许也曾风光一时，但最终还是步履维艰，甚至烟消云散。决定能否与技术趋势同行的一个核心因素就是技术架构。如果说商业模式是商业组织的灵魂，那么架构就是灵魂的根基。最典型的例子就是电商 618、双 11 大促，流量爆发之下如何保障整个商品交易流程的体验，对系统设计和运维保障都是巨大的考验，这也推动了技术的进步。当前的大促活动依然让电商企业如履如临，然而挑战难度已经不如七八年前，这就是架构演进趋于成熟的表现，充分利用云计算理念的“互联网架构”由此为业界所推崇。

综合考虑 IT 资产、业务规模、发展阶段、人才储备及投资成本，不同企业需要的互联网架构并不完全一样，譬如初创公司考虑百万级并发可能不合时宜，然而为长远计，想跟上发展迅速、爆发力惊人的互联网业务，架构设计需要既能满足当前业务需求，又具备快速升级支撑下一个发展阶段的能力。企业如何正确驾驭架构的设计和演进？有哪些通用的成熟经验可供借鉴？业界对此不乏零星的讨论，但企业仍缺乏系统的、可指导实践的参考资料。

本书是网易云基础服务架构团队根据网易集团的十余年实践，结合社区优秀案例总结而成的互联网架构演进指南。2006 年，怀着储备互联网技术和孵化新业务的初衷，网易杭州研究院扬帆起航，及至今日在两个方向上都有不俗的成绩，一端是分布式、云计算、大数据、人工智能、增强现实，一端是教育、电商、金融、游戏，这一切都是构建在千锤百炼的互联网架构之上的，这就是同时满足功能性需求、非功能性需求和产品快速迭代需求的云原生架构。网易云希望把这些经验分享给业界，与各行各业一起践行中国云计算。

全书共 5 章，以商业组织的互联网业务成长为主线，着眼于业务需求，清晰地剖析互联网架构的挑战，云原生架构的特征、构成和解决的问题，以及架构演进的路径，并通过案例对设计原则和实践加以诠释。通过阅读本书，架构师能够借鉴网易十余年的经验结晶，无须反复试验，即可快速设计符合互联网业务场景需求的架构，而有一定基础的读者朋友

也能更加透彻地理解和规划未来的方向。

全书由尧飘海统筹构思，网易云基础服务架构团队成员参与写作，书中部分图片脱胎于社区文档或官方宣传资料。第1章以宏观视角简明地解析互联网业务挑战及架构演进要点；第2章介绍开始搭建业务系统时需要的项目版本管理工具、以 Docker 为代表的容器技术，以及常见语言的工程化构建方式，并结合实战示例给予完整展现；第3章针对访问流量不是很大的业务起步阶段，讲述如何做好技术选型，实现一个支持快速迭代、高可用、安全的应用服务端；第4章介绍如何解决业务高速增长阶段的可扩展性、性能、系统集成与交互、数据可靠性等挑战；第5章以分布式定时任务和分布式锁系统的实现为例，介绍分布式服务应用挑战和架构方法，重点解析了微服务架构、分布式数据一致性和同城多活的实践。

全书内容分别由尧飘海、焦智慧、王新勇、张小刚和冯常健主笔，黄庆兵、郭忆、乔安然、何李夫、孙建良、刘发明、沈明星、崔晓晴、易庭、祝剑锋、姜政冬、朱凌墨等网易云架构师和工程师参与了写作，插图由李倩倩和纪桂莲支持完成。

本书的付梓要特别感谢电子工业出版社的符隆美编辑，她帮助我们及时地解决了遇到的各种问题。

限于作者的精力与能力，书中难免出现一些疏漏之处，请广大读者不吝指正，并予以包涵，我们会在再版中修正。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务。

- **提交勘误：**您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流：**在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31516>

二维码：



推荐序一

不知不觉，网易公司已经走到第 20 个年头。网易的业务从邮箱、门户和游戏业务，拓展到电商、文娱、教育和农业板块，并在云计算、大数据、人工智能、增强现实等领域进行了战略布局。每一项业务的健康发展，从“道”的层面而言是在于与用户共同成长，从“术”的层面则是不断打磨产品，使之更好地满足用户需求。

互联网信奉“快”，但网易更尊重“匠心”，产品每一个特性的更新，都需要经过反复的验证。这也意味着，网易业务多元化背后需要大量的产品研发和迭代工作，因此，与之匹配的应用架构至关重要。网易架构在业务成长中不断演进，形成了当下的云原生架构，让 95% 的互联网业务真正运行于云端，业务团队只需要努力为业务编码。由此，“快”和“匠心”在产品端得到了平衡。我相信，云原生是互联网业务的最优解。

云原生与传统云计算最大的区别在于，传统云计算关注的是如何提供性价比最高的计算、存储、网络资源，而云原生关注的是如何让产品能够支持快速验证业务模式，如何简化复杂的开发流程、提升研发效率，如何保障产品的高可用性让业务无需承受成长之痛，如何实现大规模弹性伸缩轻松应对业务爆发，等等。也正因如此，“云原生架构”虽然只有短短的五个字，其落地却隐藏了无数的变量与陷阱。

鉴于此，本书系统梳理了网易云原生架构的演进经验，对不同的挑战分别给出相应答案。网易架构仍在演进，我不能说这些答案已经很成熟，但我相信，书中一定会有一些章节正好击中不同企业的痛点，帮助企业在互联网业务创新过程中少走一些弯路。网易希望借此与大家共同交流探讨。

陈 刚

网易杭州研究院院长

推荐序二

作为一家诞生于中国互联网萌芽期的互联网公司，网易具有丰富的产品线。长久在网易任职，一个很大的好处是能够真切地经历波澜壮阔的互联网发展史，而对于技术人员来说，更重要的是能够学习和实践各项支撑互联网爆炸式发展的技术，能够在这段经历中获得不同于纸上得来的知识和思想：能切实体会到许多技术为何产生、为何沉寂或者兴起，从而能够在实践中避免因循守旧或一味求新逐异；更懂得如何根据功能、非功能性需求进行取舍，选择更有生命力的技术；更加理解架构的本质。

我和我的团队从十多年前的博客时代开始接触互联网技术，当时 Web2.0 概念刚刚兴起，网易博客一上线，用户量、访问量就呈爆发式增长，我们最大的困难并不在于如何编写代码，而是怎样支持产品的高速迭代。以往我们并未经历业务几乎每天都在更新迭代的情况，此时通宵更新版本成了家常便饭，为此我们优化了版本控制策略，研发并集成了自动化构建和发布工具，将其总结为“项目工程化”，在之后的项目中尤为重视。

然而，当时博客业务也在变得日益复杂。更新版本时，团队间的协调成为拖慢迭代节奏的重要因素，因此我们开始把一些业务模块独立出来，以远程接口方式提供服务，或是在负载均衡入口直接按业务模块分流，后台的缓存、数据库也做了相应的拆分，服务化进程就这样自然地开始了，团队曾考虑借鉴当时业界流行的 SOA（面向服务架构）理念，想引入企业服务总线等重量级设施，但由于 2C 业务和性能意味着用户量的支持及实际的用户体验，基于对更轻量、更高性能的渴望，我们最终选择了类似微服务的架构形态。

服务化一旦开展，过程就变得更加复杂。随着发布的频度再次提升，服务单独升级带来了版本问题，以及维护、故障期间的雪崩等一系列棘手的问题，代码质量也开始下降，由于各种方式的 RPC 调用、服务发现封装方式大量地充斥在代码中，我们意识到必须要有统一的框架支持服务化进程，于是近十年前我们有了自己的微服务框架，逐步解决雪崩、版本、服务发现、降级等问题。

后博客时代，我们将工程化、服务化等工具、框架应用于新的业务，又发现了新的问题。服务化带来了业务计算节点种类和数量的大幅增长，运维部署变得非常困难，在资源层面，服务化后的节点无法充分利用服务器资源，众多的服务被混合部署在同一台服务器上，从而服务间资源争抢，故障恢复时与各产品、各模块的协调成了最麻烦的问题，我们

意识到单台服务器的负载在短短几年内已发生了极大的改变，要解决资源管理问题，云计算势在必行。

很幸运的是我与团队能受公司之托来负责云计算平台的研发工作，2012 年秋，网易云计算平台正式开始支持公司业务，得益于 SDN 技术与公司原内网环境的较好融合，云平台很快得到了广泛使用。相比物理机节点数量的线性增长，虚拟机的数量指数级增长在很大程度上解决了原本遇到的资源管理问题；虚拟主机做到专机专用按需配置，使用数据库、缓存等基础服务再也不用等待运维团队部署维护，系统运维效率也得到了极大的提升。

获取资源容易了，产品自然而然规划出众多的测试环境，如开发环境、集成环境、预发布环境等，于是搭建测试环境，持续集成与交付很快成了新瓶颈，更麻烦的是业务服务化架构日渐成型，应用运维的复杂度指数级提升。为此，我们又打造了自动化部署平台，解决集群的编排、版本更新、回滚等问题，平台上线后每月的部署次数直线上升，达到数万次之多。没有云计算时，我们觉得两套测试环境共用就行，每天集成部署一两次就行，有了云计算后，原本被压抑的资源、迭代频度的需求被瞬间释放，反应到产品中的便是功能开发并行度和迭代速度的大幅提升，迭代风险的大幅下降。上云的收益由此可见一斑，先行拥抱云计算的企业在产品竞争中可获得巨大的优势。

可以说在网易这样的成熟互联网公司，是原本的软件架构、技术体系的进化推动了云计算的实施，塑造了云计算平台的形态，我们的业务技术架构是与云计算伴随生长的，并且经历了漫长的迭代过程，而对于后来者，基于云来设计软件架构、实现工程化、建设运维体系，则可以选择一条前人已充分实践并总结和提炼的路径，一开始便可以从云而生，这也就是我们所说的云原生的技术体系。我们在开放网易云计算能力的过程中遇到过不少用户，他们不知道网易这样的互联网企业如何基于云高速迭代产品、支撑海量用户，不知道如何在架构设计、技术选型阶段迈出第一步，如何为未来的发展打下基础，或是业务增长后如何应对。有初创企业虽然灵活快速却控制不了迭代质量、刹不住车的，也有传统企业提不了速的。正是看到这些问题，我认为，作为互联网技术的实践者，我们不仅应当将我们的平台、工具链开放出来，同时也应将我们的云端架构实践梳理成知识体系分享出来，在这“大众创业，万众创新”的互联网+时代为企业健康发展作出力所能及的贡献。

陈 谔

网易杭州研究院云计算技术部总监

目 录

| | |
|--------------------------|----|
| 引子 | 1 |
| 第 1 章 互联网系统架构的挑战 | 2 |
| 1.1 云应用架构技术发展 | 4 |
| 1.2 云平台下架构的不同点 | 5 |
| 1.2.1 开发模式的区别 | 6 |
| 1.2.2 交付模式的区别 | 7 |
| 1.2.3 架构设计的区别 | 8 |
| 1.3 云原生应用架构 | 10 |
| 1.4 架构演化发展历程 | 21 |
| 1.4.1 初创期架构 | 22 |
| 1.4.2 快速成长期架构 | 24 |
| 1.4.3 分布式服务架构 | 26 |
| 1.5 云计算服务介绍 | 29 |
| 1.6 云计算解决方案 | 31 |
| 1.7 案例概述 | 34 |
| 1.7.1 背景介绍 | 34 |
| 1.7.2 环境要求 | 36 |
| 1.7.3 项目构建 | 36 |
| 1.7.4 项目运行 | 36 |
| 1.7.5 相关技术介绍 | 37 |
| 小结 | 40 |
| 第 2 章 从 0 到 1 工程实践 | 41 |
| 2.1 工程化 | 41 |
| 2.1.1 工程模板 | 41 |

| | | |
|-------|--------------|-----|
| 2.1.2 | 模块化 | 45 |
| 2.1.3 | 工程化构建 | 50 |
| 2.1.4 | 代码规范及检查 | 53 |
| 2.1.5 | 代码版本管理 | 54 |
| 2.1.6 | 环境划分 | 61 |
| 2.2 | 基于容器工程化 | 62 |
| 2.2.1 | Docker 及作用 | 63 |
| 2.2.2 | Docker 镜像及操作 | 66 |
| 2.2.3 | Docker 容器及操作 | 73 |
| 2.2.4 | 基于容器工程化 | 77 |
| 2.3 | 实战示例 | 78 |
| | 小结 | 84 |
| 第 3 章 | 初创期应用架构实践 | 85 |
| 3.1 | 技术选型 | 85 |
| 3.1.1 | 业务框架选型 | 85 |
| 3.1.2 | 结构化数据存储 | 92 |
| 3.1.3 | 缓存选型 | 102 |
| 3.1.4 | 静态资源存储 | 106 |
| 3.2 | 架构实践 | 109 |
| 3.2.1 | 快速迭代 | 109 |
| 3.2.2 | 高可用与负载均衡 | 111 |
| 3.2.3 | 交付与部署 | 117 |
| 3.2.4 | Web 应用安全 | 119 |
| 3.3 | 应用监控 | 127 |
| 3.3.1 | 应用监控指标 | 127 |
| 3.3.2 | 应用进程监控 | 128 |
| 3.3.3 | 操作系统监控 | 129 |
| | 小结 | 136 |

| | |
|-----------------------|-----|
| 第4章 快速成长期应用架构实践 | 137 |
| 4.1 关键业务需求 | 137 |
| 4.1.1 计数与排序 | 137 |
| 4.1.2 秒杀 | 146 |
| 4.1.3 全文检索 | 149 |
| 4.1.4 日志收集 | 154 |
| 4.2 架构实践 | 156 |
| 4.2.1 前端系统扩展 | 157 |
| 4.2.2 无状态服务设计 | 157 |
| 4.2.3 在线水平扩展 | 160 |
| 4.2.4 后端系统扩展 | 163 |
| 4.2.5 系统通信 | 173 |
| 4.2.6 消息中间件 | 176 |
| 4.3 系统优化 | 181 |
| 4.3.1 静态资源分离 | 182 |
| 4.3.2 数据库调优 | 185 |
| 4.3.3 系统高可用 | 193 |
| 4.4 应用诊断 | 200 |
| 4.4.1 应用健康检查 | 200 |
| 4.4.2 性能问题诊断 | 204 |
| 4.4.3 基于日志的故障诊断 | 210 |
| 4.5 数据库诊断 | 214 |
| 4.6 DevOps | 223 |
| 4.6.1 持续集成 | 224 |
| 4.6.2 持续交付 | 227 |
| 4.6.3 灰度发布 | 229 |
| 4.6.4 大应用编排 | 231 |
| 4.7 安全设计 | 246 |
| 4.7.1 入侵检测 | 247 |
| 4.7.2 防劫持攻击 | 249 |

| | |
|---------------------|-----|
| 小结 | 255 |
| 第 5 章 稳定期服务化应用架构实践 | 256 |
| 5.1 业务拆分 | 256 |
| 5.2 统一配置中心 | 259 |
| 5.3 分布式定时任务 | 261 |
| 5.3.1 分布式定时任务设计 | 262 |
| 5.3.2 业界流行的开源框架 | 264 |
| 5.4 分布式锁系统 | 274 |
| 5.5 微服务化架构 | 277 |
| 5.5.1 服务发现 | 279 |
| 5.5.2 服务治理 | 302 |
| 5.5.3 微服务框架 | 307 |
| 5.5.4 服务编排 | 313 |
| 5.5.5 微服务测试 | 321 |
| 5.6 分布式数据一致性 | 333 |
| 5.6.1 CAP 和 BASE 理论 | 333 |
| 5.6.2 一致性模型 | 336 |
| 5.6.3 典型的解决方案 | 337 |
| 5.7 同城多活 | 344 |
| 5.7.1 应用同城多活 | 345 |
| 5.7.2 跨 AZ 负载均衡 | 347 |
| 5.8 故障诊断 | 348 |
| 小结 | 353 |
| 参考文献 | 354 |
| 技术术语 | 356 |

引子

Netscape 创始人、硅谷著名投资人马克·安德森 (Marc Andreessen) 6 年前在《华尔街日报》上发表了“软件正在吞噬整个世界”的论断。目前来看，这一论断得到了有力的证明，软件的发展趋势越来越强劲。当前世界，每时每刻都有软件在发布和更新。特别是随着移动互联网技术的发展和普及，软件也渐渐变成我们生活中的一部分，从而对我们的生活方式产生非常重大的影响。然而，如此大数量和大规模的软件开发依赖非常多的人力和时间去设计、开发、测试和发布，因此，如何在满足产品发布需求的情况下，保证软件高效的生命周期管理，需要考虑的因素非常复杂，大部分通过分层抽象，分解成一个个小问题来处理，云原生应用就为解决此类问题提供了一种有效的方法。

首先，由于产品快速迭代的需求，高效组织或个人一般会通过云原生服务来快速构建和交付可用性更高的应用，云原生应用由于天然基于云服务的开发模式，通常包括持续集成、DevOps 和微服务架构等过程。毫无疑问，这些过程是相互融合、交织在一起的，只能通过自动化来提高整体的运行效率。所以成员之间相互信任、主动快速、良好的操作体验也可以在云原生应用中得到体现。

其次，软件系统一定会发生错误，云原生应用通过不同的服务来构建可靠的软件系统，因此为错误而进行架构设计必须提供自助式服务，服务架构的设计需要满足可伸缩性并具有容错性，软件的出错后行为应该是可预测的，应用通过自动恢复来保证业务的可用性。

最后，云原生应用使得用户把焦点更多关注在代码及架构本身，通过完全自动化模式来交付软件，以应用为中心，而不用关注后台是在和虚拟机还是容器或者其他基础设施打交道。因此，云原生应用设计时需要从特定的基础设施中进行解耦，不能与特定的平台绑定，依赖于团队的架构设计能力和协作约定，以工程化、服务化和自动化的流程来进行应用开发。

第 1 章 互联网系统架构的挑战

21 世纪初期，随着互联网接入逐渐普及，Web 2.0 业务模式飞速发展，大量互联网或者企业应用系统所需要处理的数据量急速增长，如何在用户数量迅速增长的情况下快速扩展原有系统，是技术人员必须面对的问题。无论是在软件资源的时间维度上，还是在硬件资源的空间维度上，资源的有限性，比如交付周期、电力成本、空间成本和维护成本等，会直接导致数据中心成本上升，而另一方面资源的使用率又不高。因此，如何在满足业务快速增长需求的同时，有效地解决成本与效率的矛盾，是企业面临的一个重大问题。借鉴传统水电煤的商业发展模式，技术人员通过集中提供资源的方式，将计算、网络和存储等资源提供集中化统一管理，于是云计算应运而生。

从 AWS CEO 安迪·加西（Andy Jassy）于 2003 年提交的商业计划书，到 2006 年 ECS（弹性云服务器）正式上线提供商业服务，再到 2016 年“AWS re:Invent 2016”大会的发展，云计算首先是在商业模式上带来了一个崭新的、获取计算资源的模式，以提供水电煤的方式来为用户提供资源，是一种对资源获取方式的变革，也是人类思维的基础和模式上的飞跃。因此，从这个意义上讲，云计算和传统主机业务相比有非常大的不同。比如，资源的使用由传统的准备订购模式变成了订阅的模式，也就是说业务需要多少资源，就可以从服务商那里购买多少资源，用户不需要过多关注资源的基础准备，这和普通的资源准备有很大的不同。同样，对资源的付费方式，也产生了多种灵活的模式，用多少就付多少费，用户可以对于固定的资源直接采用“包年包月”的方式，也可以按需付费，甚至可以做到的在不同的时间段采取不同的收费模式，这和我们日常使用峰谷电的模式思路类似，都是为了实现资源利用率的最大化。另外，用户也无须在建设或租用机房、网络规划等方面投入大量的人力和硬件。

在“大众创业、万众创新”的政府政策和市场环境下，互联网化需求逐渐明显，伴随着互联网技术日新月异的发展和创业门槛的日益变低，从早期的独角兽到如今的大众创业，云化在国内外是个再普通不过的话题了。每个创业公司不再是讨论要不要上云，而是讨论

如何上云及如何快速上云。同样，技术的发展也带来了传统行业的冲击，移动互联网的飞速发展，智能手机的普及，每个人的碎片时间越来越多，造就了越来越多的机会，特别是大型互联网公司与传统企业的结合带动了更快的消费升级。

传统行业对互联网的转型需求也越来越大，很多大型传统国有企业、金融和银行业都在积极响应国家的号召，争取大部分业务上云。一方面是新型的创业公司根据业务发展不断调整系统架构，另一方面是传统行业的互联网转型将面临新业务的建立和遗留系统的集成。

在项目初期，云计算为用户带来的好处显而易见。首先，互联网项目前期资金投入在固定资产的比例越小，对产品开发或者创业来讲成本风险就越小，从而有更多资源放在业务上，云计算把计算资源从以前昂贵的预备方式变成一种使用较低的成本就唾手可得的方方式；其次，使用云计算以前，计算资源的获取往往是一个比较漫长的过程，涉及选型、下单、生产、运输、初始化再到环境的搭建等多个环节，而现在变成了云计算公司的选择、数据中心的选择、云主机的选购，以及服务器的配置、安装、调试等过程。新的方式变成了一个简简单单的订购过程。如果是自己建设，除去各种政策的影响，仅机房的位置选择就是一个非常漫长的过程。而现在，很多情况下，只要在云计算公司的网站上像选择普通商品一样，加入购物车并支付，就可以马上获取所需资源，如果发生错误或者不需要了，可以立即销毁重做，不需要涉及复杂的下架回收流程。云计算缩短了很多产品的发布周期，这对于中小公司特别是刚刚起步的公司来说，大大提高了工作效率。

互联网的规模不仅越来越大，而且随着移动互联网的出现和飞速发展，热点随时随地可能产生。比如一些热销产品，一个热门事件就有可能导致系统流量的暴涨，特别是如今名人、大V等的宣传或转发，对系统的压力要求更大。比如之前的某明星离婚事件、某电子商务运营秒杀抢购业务等。

当然，正是由于初创公司不同业务的快速发展，才有机会使得互联网从业人员有更多的机会去挑战不同的技术难题，使用更好的技术以更好地服务用户。通常，很多业务型企业一开始都不太注重基础设施建设，也比较少有机会对系统进行良好的架构设计，随着企业的飞速发展，往往会碰到各种技术问题，可能会影响产品的迭代速度，容易错失很多机会，如果这时候对产品的核心架构进行调整，就像为高速行驶的汽车更换引擎一样，是一件很困难的事。诚然，这种比喻有点夸张，但是对于竞争激烈的企业来说，这种机会就显得非常重要，直接采用云服务的模式来打造，架构设计会更简单，同时还能提供更高的可

用性保证，也不用非常大的基础投入，算是云服务给架构设计带来的一大优势。

因此，满足产品的快速迭代需求，保证系统既能满足功能性需求又能满足非功能性需求，是技术人员所追求的目标，也是技术的价值所在。

1.1 云应用架构技术发展

随着云计算的发展和进化，以及企业对效率的追求变得越来越高，大量的数据表明，简单的云主机创建也不太能满足业务的需求，后续还有大量的运维和运营工作，运维操作频率基本占比在 90% 以上，尤其在业务本身不断发展并且规模不断扩大的时候会更加明显，矛盾也会越来越突出。所以，如何在速度与稳定性之间追求平衡，达到效率的最优化是本阶段用户追求的目标，因此我们迫切需要新一代的云计算技术出现，使得用户有更大的聚集业务，而不是烦琐的重复机械操作，在服务发现、编排和集成方面有完善的自动化工具集来帮助用户代替手工操作，帮助技术人员发现问题、解决问题、预测问题，甚至自动修复问题。

显然，商业模式和技术模式都在不断变化，互相促进，对开发者来讲，基于云计算的开发模式自然会导致思维的转变。当然不是说老的模式就不可以使用，而是说如果要更好地发挥云的特性，就需要在开发模式上进行最佳实践，尤其需要在架构设计上进行适当的调整。比如对使用资源的考虑，必须从单一服务器的资源，逐渐向整个资源池或数据中心的方向转变，架构也需要从单体的架构向服务化的架构转变。相应地，在架构设计的时候，面向的是整体资源的需求及运用，而不是在设计架构时局限于某一台物理服务器，因为所有的服务都有可能失败，也是不可靠的，错误也无法避免，如果以服务器为单位来进行架构设计就变得不适合了，这是一个比较大的变化；另外，从自建模式变成租用模式，很多人会对安全性有顾虑，认为自建更可靠，实际上这是个伪命题，类似认为把钱放在自家比放在银行更保险一样，实际上无论放在哪里都可能发生问题，因此，要根据用户的数据本身对安全风险的承受度，来重新考虑数据的安全问题。如果想降低风险，可以把数据做更细小的划分，划分的数据如何分别存储和调度，是需要设计的。因此需要开发模式的变化，把云计算的基础设施当成代码去使用，融合到设计中去。

作为基础架构的提供商，老一代云计算的服务提供商像水电力公司，新一代的云计算服务商更像现代化的工厂。互联网业务的发展和工业的发展是相辅相成的，比如工业 4.0

的理念与实践，目的之一是实现工业从集中式向分布式的发展方向，和架构演变模式完全一致。

简而言之，用户最需要云服务实现以下 3 点基本特性，第一是可用性，服务商随时保证对服务的可使用性，就像在家需要随时能够打开电源开关，随时能够取用水资源；第二是可扩展性，也叫弹性，服务商能够随时满足业务在爆发时的需求，就像家里有朋友集会时，我可能需要更多的水，更多的电源，基础服务公司需要去负责这些资源的调度，而用户不需要去做其他改变；第三是可管理性，用户能够随时对基础设施提供的服务进行管理，自动控制服务，就像装修时布线、开关和插座等的设计，需要能够满足不同房间的各种不同需求，比如书房和儿童房的需求是有区别的。

对于云计算服务商而言，这 3 点都是必不可少的要素，是必要条件，只有满足了这 3 个条件，才能够满足最终用户的基本需求。所以为了提供更多的优质服务，需要新的云计算服务，需要 IaaS 服务和 PaaS 服务最终融合在一起，提供更流畅的、一致的生态体验服务。无论提供怎样的方式、提供何种类型的服务，对于最终用户来说，是追求效率的提高和资源的最大化利用。总之，用户对于云计算服务使用要求的多样性，要远远比任何一种以前的传统服务多得多，基础设施的可编程性和服务的 API，对于云计算厂商来说都是非常重要的。

云计算技术的发展刚好见证了互联网技术的发展，从物理机、虚拟机到容器技术的使用，从传统的瀑布开发到敏捷流程再到精益迭代，互联网技术对开发者的要求也会越来越多样化，甚至包括多地理位置的多样协作等，为开发者带来更多的问题同时，也会带来许多有意思的技术内容，既是机遇也是挑战。

1.2 云平台下架构的不同点

毫无疑问，现在是云计算时代，前文也阐述了云计算不仅是一种 IT 资源的使用方式，而且已经演变成一种生态，它不仅会对软件开发人员的开发流程带来一定改变，更重要的是会为软件架构设计带来深刻影响。二者相互结合，也将导致对整个产品的上线和迭代速度产生连锁反应，比如云计算使得开发者无须再关注硬件资源，使得交付流程更快。然而，又可能由于依赖云服务厂商的技术稳定性不一样，所以在架构设计的时候要更多考虑到云的特性来满足非功能需求。2013 年微服务的呈现到现在非常火热的场景刚好和云计算的发

展一致，有些人甚至认为它就是云服务时代的产物。如何基于“云”做软件架构设计，或者如何让自己原来的软件架构能迁移到“云”上，本书给出了原理和初步的方法，相信能给读者带来一些启示，我们先来总结一下云计算对技术人员的影响。

1.2.1 开发模式的区别

开发人员和测试人员之间

传统的开发人员在本地开发或测试完成之后，还需要重新上传到服务器端进行环境的安装、搭建和调试，在最后上线时又要重新做一次之前的工作，整个流程不仅烦琐，而且容易出错；除此之外，测试人员也会对测试的内容质量没有完全的把握，因此不能完全保证所有的运行环境完全一致，容易产生沟通矛盾等问题。

开发人员和运维人员之间

一般来说，产品经过开发测试之后，常常会分发到各阶段流程中，即产品上线。上线的流程通常包括资源申请、系统安装、环境配置等，有些可能包括容量规划、资源申请与审批、工单申请和反复沟通等，即使是在使用虚拟机的阶段，也需要有一个过程。根据权威的调查报告，一般虚拟机的创建时间最长不超过 15 分钟，然而业务上线却要长达 2 到 3 周，整个流程非常缓慢。

图 1-1 中，基于物理的开发流程要经过业务需求、采购审批、安装机器、软件安装到应用部署 5 个步骤，基于虚拟机的开发流程只要经过业务需求、软件安装和应用部署 3 个步骤，相对来说流程有几步优化。但是实际上由于软件安排、升级的依赖性，团队仍然需要大量的时间去做基础的重复劳动，效率提升不明显。基于以应用为中心的模式只需要从业务需求到应用部署 2 个步骤，把第二种开发流程中烦琐的软件安装通过自动化的模式来完成，保证了软件环境一致性，减少了系统依赖的风险，同时开发人员只要聚集在应用上，其他由基础设施服务一键完成，大大提高了软件的生产效率。

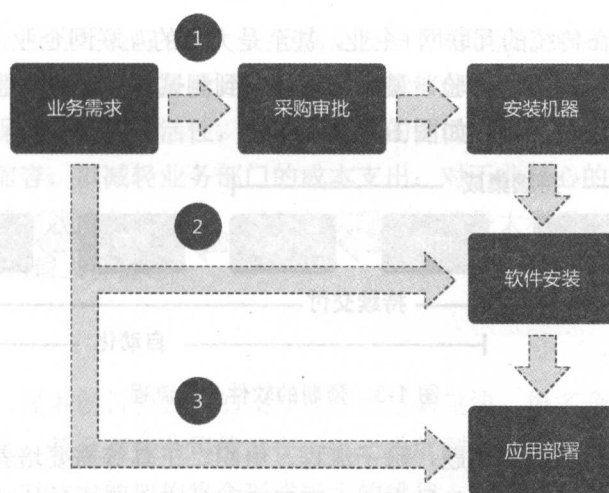


图 1-1 3 种软件开发流程比较

1.2.2 交付模式的区别

开发人员或者架构人员在完成应用的开发测试后，还要编写对应的文档来和运维人员描述系统的部署架构，但是开发人员和运维人员对技术的理解存在不一致，同时对业务的理解也不一样。我们不可能让运维人员重新学习相关的业务知识，让他们完整理解整个业务的架构又相当困难，毕竟运维人员不像开发人员一样能理解整个产品的需求和开发过程，而且他们同时还可能兼任各个产品的运维任务，没有那么多精力来完全了解整个业务架构。因此整个发布流程是中断、有隔离的，并不流畅，如图 1-2 所示。

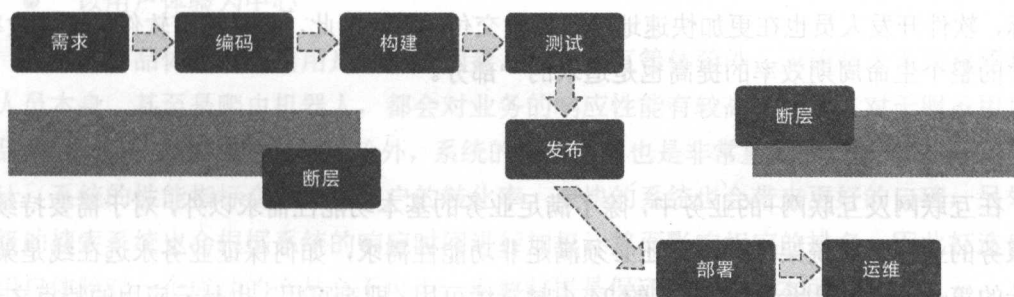


图 1-2 软件发布流程

除此之外，无论大小公司，还存在运维人员人力招聘、成本预算等问题，由于这个问题和本书的主题关系不大，对应的背景和上下文在此就不再详述。总之，无论在创业型的

互联网企业，还是在传统的互联网+企业，甚至是大型的互联网企业，都希望能够进行比较流畅的开发测试上线的流程体验。最后，从开发到测试再到上线及监控反馈，整个过程应该能够不断反馈和螺旋上升，如图 1-3 所示。

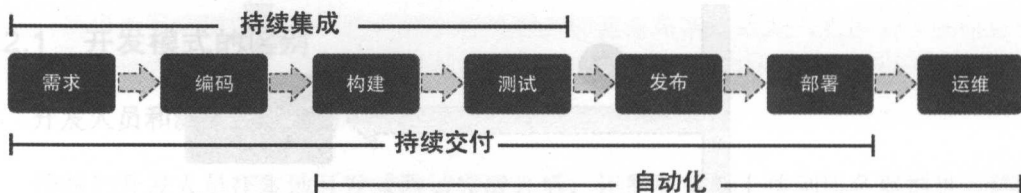


图 1-3 预期的软件迭代流程

因此，为了解决相关的问题，除了流程、组织、工具等需要培养之外，既需要技术人员在实现软件开发和交付模式的转变，也更需要在技术上关注架构的调整来满足软件的架构需求作保证。

1.2.3 架构设计的区别

云应用架构设计意味着更快的迭代速度、持续可用的服务、弹性扩容及一些非功能需求，包括追求产品创新时间的技术挑战、以用户体验为中心的挑战和移动互联网时代的突发性挑战。

更快的迭代速度

无论是个人还是企业，抓住更多的商业机会、更加快速地交付软件是整个组织追求的目标，软件开发人员也在更加快速地迭代持续交付软件，因此，除了提高软件的开发效率，软件的整个生命周期效率的提高也是追求的一部分。

持续可用的服务

在互联网及互联网+的业务中，除了满足业务的基本功能性需求以外，对于需要持续提供服务的业务，系统架构的设计还必须满足非功能性需求，如何保证业务永远在线是架构设计的第一要素，因此，实现业务的 24 小时持续可用（即高可用）也是云应用的特点之一。

弹性伸缩

病毒效应在互联网场景下非常明显，因此，能够根据业务的流量和系统压力进行智能

的弹性计算，也是云应用的显著特点之一，特别是在面对流量爆发或者突发事情的情况，还能保证业务的可用性，因此，业务设计的时候是否能够分析出业务的关键路径和热点部件，同时在架构层面保持足够的灵活性，以满足业务的弹性要求，并在低压力的情况下，保证资源的占用自动缩容，以减轻业务部门的成本支出；对于非核心的业务，启用避开峰值的方式来实现在线或离线业务的计算，尽可能实现云计算最大利用率，也就是常说的用好“云”，发挥云计算的最大价值。

非功能需求

随着技术的发展，很多创新型公司的发展速度也越来越快，很多企业能够在较短的时间内达到独角兽的角色，甚至有些公司能够在一年之内就到达上千万用户量的规模。在业务急速发展的过程中，对技术的架构将会形成极大的挑战，如果从一开始没有很好的技术架构支撑，将导致后续的技术债越来越重，严重影响业务发展。除了业务对技术的影响外，技术也会对业务带来影响，两者相辅相成。这类创业公司常常具有很多共同的特征，例如追求产品创新和极致的用户体验，主要包括以下几部分。

● 云模式的产品创新

现在每年有非常多的企业诞生，很多企业都希望把传统行业和互联网结合起来追求新的商业模式，所以在不停地进行产品研究和创新，对此类企业而言，时间是一个非常重要的因素。此外，产品上的改变也越来越多地依赖“云”上的技术来完成，比如人工智能、数据推荐等技术，这些产品的创新必须依赖云计算的技术才能完成。

● 以用户体验为中心

良好的产品体验是吸引用户的首要因素，除去交互等体验外，无论是个人用户还是企业人员本身，甚至是爬虫机器人，都会对业务的响应性能有较高的要求。对于服务用户的产品来说，除了人性化的交付体验外，系统的响应时间也是非常重要的。根据 eBay 之前的统计，系统的性能指标会影响到用户的转化率，更快的系统也会带来更好的口碑，另外，高级的搜索系统也会根据系统的响应时间进行加权计算而影响相应的排名，因此打造良好的用户体验对一个成功的产品必不可少，当然前提是保证产品的基本可用。

● 移动互联网的突发性

随着移动互联网的技术发展，越来越多的人依赖于移动终端而生活，因此对业务的可用性提出了更高的要求，碎片化的时间使得用户能随时随地进行网络互联操作。另一方面，

在节假日或者热点事件发生时，必须随时应对用户的突发访问压力，因此提高系统的非功能性要求也给技术人员在架构设计方面提出了新的挑战，相应地，也能为新业务提供更多的机会。

很明显，仅仅考虑快速验证和成本的投入因素，传统的软件生产方式根本无法满足产品对技术的发展要求，因此，使用“云”是目前所有公司的首要选择方案之一。然而，如果只是以资源为对象的方式来使用“云”，而不是以应用为中心、聚集于业务，将会导致效率非常低下，仍然无法满足用户快速迭代等需求，因此需要根据“云”的特点从架构上来进行设计，所以基于“云”的应用架构是所有技术人员从事开发必须考虑的首要条件。如何基于云计算平台设计出具有高扩展、高可用的架构来满足业务需求，创造更大的商业价值，也是原生应用架构设计的必经之路。那么，何为云原生应用？我们来一探究竟。

1.3 云原生应用架构

云原生（Cloud Native）的概念，由来自 Pivotal 的 Matt Stine 于 2013 年首次提出，被一直延续使用至今。这个概念是 Matt Stine 根据其多年的架构和咨询经验总结出来的一个思想集合，并得到了社区的不断完善，内容非常多，包括 DevOps、持续交付（Continuous Delivery）、微服务（MicroServices）、敏捷基础设施（Agile Infrastructure）和 12 要素（The Twelve-Factor App）等几大主题，不但包括根据业务能力对公司进行文化、组织架构的重组与建设，也包括方法论与原则，还有具体的操作工具。采用基于云原生的技术和管理方法，可以更好地把业务生于“云”或迁移到云平台，从而享受“云”的高效和持续的服务能力。

顾名思义，云原生是面向“云”而设计的应用，因此技术部分依赖于在传统云计算的 3 层概念（基础设施即服务（IaaS）、平台即服务（PaaS）和软件即服务（SaaS）），例如，敏捷的不可变基础设施交付类似于 IaaS，用来提供计算网络存储等基础资源，这些资源是可编程且不可变的，直接通过 API 可以对外提供服务；有些应用通过 PaaS 服务本来就能组合成不同的业务能力，不一定需要从头开始建设；还有一些软件只需要“云”的资源就能直接运行起来为云用户提供服务，即 SaaS 能力，用户直接面对的就是原生的应用。

应用基于云服务进行架构设计，对技术人员的要求更高，除了对业务场景的考虑外，对隔离故障、容错、自动恢复等非功能需求会考虑更多。借助云服务提供的能力也能实现更优雅的设计，比如弹性资源的需求、跨机房的高可用、11 个 9（99.999999999%）的数据

可靠性等特性，基本是云计算服务本身就提供的能力，开发者直接选择对应的服务即可，一般不需要过多考虑本身机房的问题。如果架构设计本身又能支持多云的设计，可用性会进一步提高，比如 Netflix 能处理在 AWS 的某个机房无法正常工作的情况，还能为用户提供服务，这就是“云”带来的魔力，当然，云也会带来更多的隔离等问题。如图 1-4 所示，目前业界公认的云原生主要包括以下几个层面的内容。

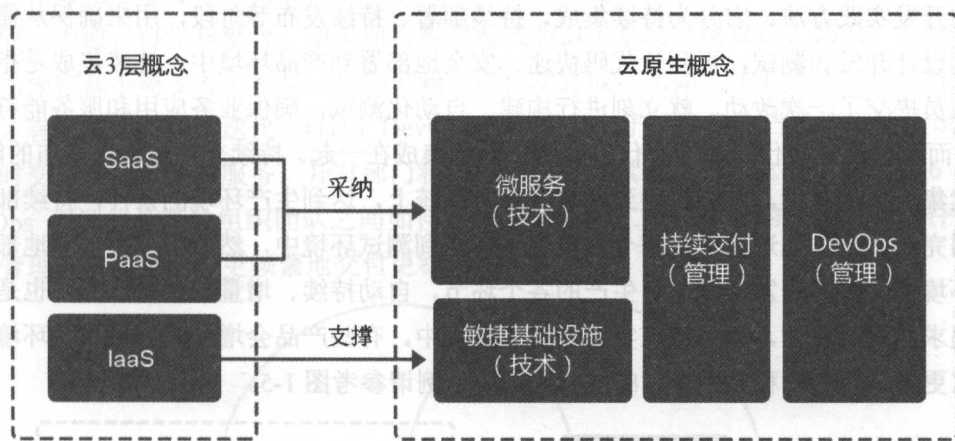


图 1-4 云原生的内容

敏捷基础设施

正如通过业务代码能够实现产品需求、通过版本化的管理能够保证业务的快速变更，基于云计算的开发模式也要考虑如何保证基础资源的提供能够根据代码自动实现需求，并实现记录变更，保证环境的一致性。使用软件工程中的原则、实践和工具来提供基础资源的生命周期管理，这意味着工作人员可以更频繁地构建更强可控或更稳定的基础设施，开发人员可以随时拉取一套基础设施来服务于开发、测试、联调和灰度上线等需求。当然，同时要求业务开发具有较好的架构设计，不需要依赖本地数据进行持久化，所有的资源都是可以随时拉起，随时释放，同时以 API 的方式提供弹性、按需的计算、存储能力。

技术人员部署服务器、管理服务器模板、更新服务器和定义基础设施的模式都是通过代码来完成的，并且是自动化的，不能通过手工安装或克隆的方式来管理服务器资源，运维人员和开发人员一起以资源配置的应用代码为中心，不再是一台台机器。基础设施通过代码来进行更改、测试，在每次变更后执行测试的自动化流程中，确保能维护稳定的基础设施服务。

此外，基础设施的范围也会更加广泛，不仅包括机器，还包括不同的机柜或交换机、同城多机房、异地多机房等，这些内容也会在后续章节中逐一进行部分讨论。

持续交付

为了满足业务需求频繁变动，通过快速迭代，产品能做到随时都能发布的能力，是一系列的开发实践方法。它分为持续集成、持续部署、持续发布等阶段，用来确保从需求的提出到设计开发和测试，再到让代码快速、安全地部署到产品环境中。持续集成是指每当开发人员提交了一次改动，就立刻进行构建、自动化测试，确保业务应用和服务能符合预期，从而可以确定新代码和原有代码能否正确地集成在一起。持续交付是软件发布的能力，在持续集成完成之后，能够提供到预发布之类系统上，达到生产环境的条件，持续部署是指使用完全的自动化过程来把每个变更自动提交到测试环境中，然后将应用安全地部署到产品环境中，打通开发、测试、生产的各个环节，自动持续、增量地交付产品，也是大量产品追求的最终目的，当然，在实际运行的过程中，有些产品会增加灰度发布等环境。总之，它更多是代表一种软件交付的能力，过程示例请参考图 1-5。

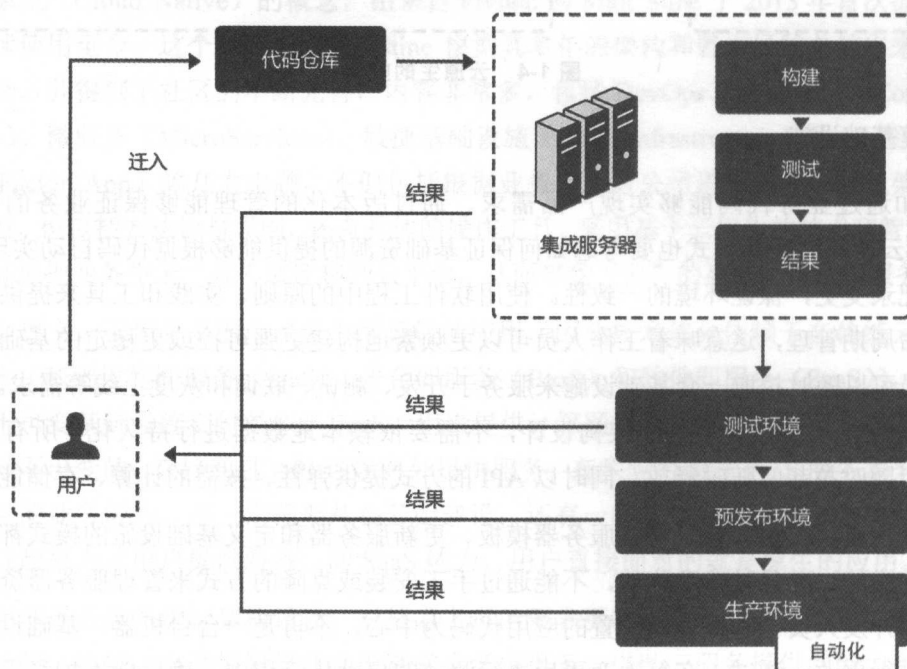


图 1-5 持续交付流程

DevOps

DevOps 如果从字面上来理解只是 Dev（开发人员）+Ops（运维人员），实际上，它是一组过程、方法与系统的统称，其概念从 2009 年首次提出发展到现在，内容也非常丰富，有理论也有实践，包括组织文化、自动化、精益、反馈和分享等不同方面。首先，组织架构、企业文化与理念等，需要自上而下设计，用于促进开发部门、运维部门和质量保障部门之间的沟通、协作与整合，简单而言组织形式类似于系统分层设计。其次，自动化是指所有的操作都不需要人工参与，全部依赖系统自动完成，比如上述的持续交付过程必须自动化才有可能完成快速迭代。再次，DevOps 的出现是由于软件行业日益清晰地认识到，为了按时交付软件产品和服务，开发部门和运维部门必须紧密合作。总之，如图 1-6 所示，DevOps 强调的是高效组织团队之间如何通过自动化的工具协作和沟通来完成软件的生命周期管理，从而更快、更频繁地交付更稳定的软件。

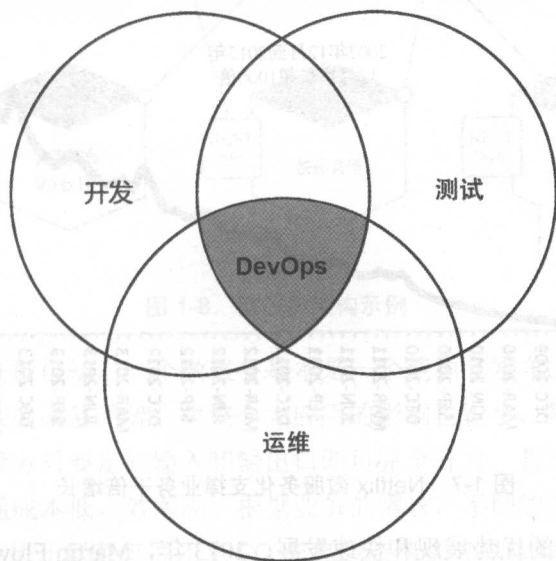


图 1-6 DevOps 强调组织的沟通与协作

微服务

随着企业的业务发展，传统业务架构面临着很多问题。其一，单体架构在需求越来越多的时候无法满足其变更要求，开发人员对大量代码的变更会越来越困难，同时也无法很好地评估风险，所以迭代速度慢；其二，系统经常会因为某处业务的瓶颈导致整个业务瘫

疾，架构无法扩展，木桶效应严重，无法满足业务的可用性要求；最后，整体组织效率低下，无法很好地利用资源，存在大量的浪费。因此，组织迫切需要进行变革。随着大量开源技术的成熟和云计算的发展，服务化的改造应运而生，不同的架构设计风格随之涌现，最有代表性的是 Netflix 公司，它是国外最早基于云进行服务化架构改造的公司，2008 年因为全站瘫痪被迫停业 3 天后，它痛下决心改造，经过将近 10 年的努力，实现了从单架构到微服务全球化的变迁，满足了业务的千倍增长（如图 1-7 所示），并产生了一系列的最佳实践。



图 1-7 Netflix 微服务化支撑业务千倍增长

随着微服务化架构的优势展现和快速发展，2013 年，Martin Flower 对微服务概念进行了比较系统的理论阐述，总结了相关的技术特征。首先，微服务是一种架构风格，也是一种服务；其次，微服务的颗粒比较小，一个大型复杂软件应用由多个微服务组成，比如 Netflix 目前由 500 多个的微服务组成；最后，它采用 UNIX 设计的哲学，每种服务只做一件事，是一种松耦合的能够被独立开发和部署的无状态化服务（独立扩展、升级和可替换）。微服务架构如图 1-8 所示。

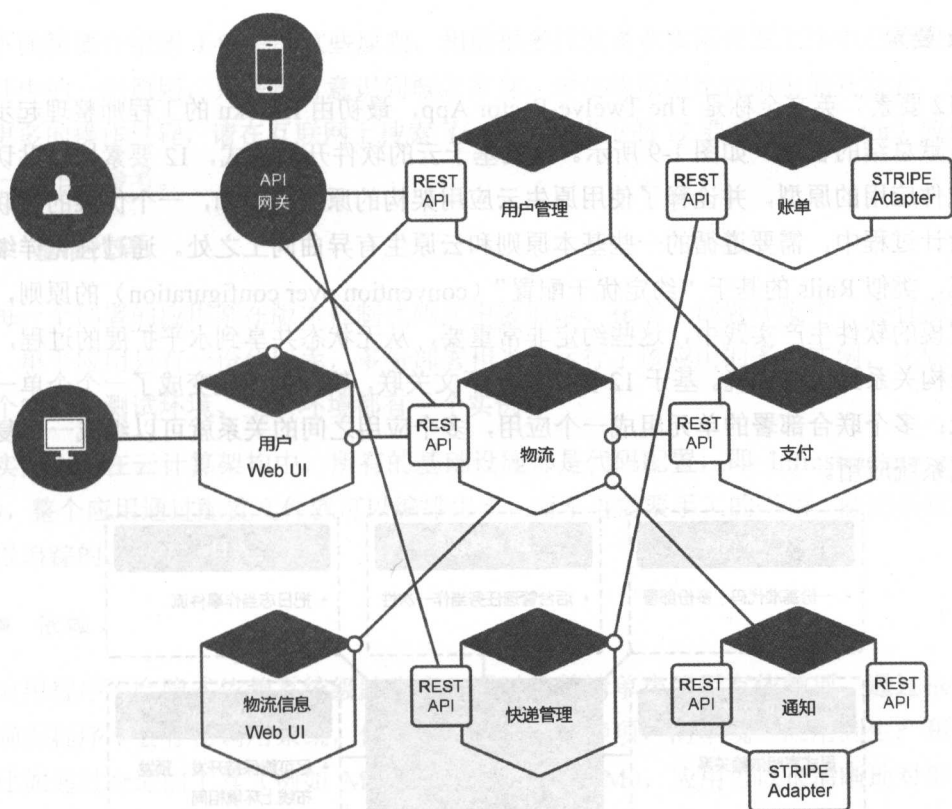


图 1-8 微服务架构示例

由微服务的定义分析可知，一个微服务基本是一个能独立发布的应用服务，因此可以作为独立组件升级、灰度或复用等，对整个大应用的影响也较小，每个服务可以由专门的组织来单独完成，依赖方只要定好输入和输出口即可完全开发，甚至整个团队的组织架构也会更精简，因此沟通成本低、效率高。根据业务的需求，不同的服务可以根据业务特性进行不同的技术选型，是计算密集型还是 I/O 密集型应用都可以依赖不同的语言编程模型，各团队可以根据本身的特色独自运作。服务在压力较大时，也可以有更多容错或限流服务。

微服务架构确实有很多吸引人的地方，然而它的引入也是有成本的，它并不是银弹，使用它会引入更多技术挑战，比如性能延迟、分布式事务、集成测试、故障诊断等方面，企业需要根据业务的不同的阶段进行合理的引入，不能完全为了微服务而“微服务”，本书第 5 章也会对如何解决这些问题提供不同的方案。

12 要素

“12 要素”英文全称是 The Twelve-Factor App，最初由 Heroku 的工程师整理起步，是集体贡献总结的智慧，如图 1-9 所示。根据基于云的软件开发模式，12 要素比较贴切地描述了软件应用的原型，并诠释了使用原生云应用架构的原因。比如，一个优雅的互联网应用在设计过程中，需要遵循的一些基本原则和云原生有异曲同工之处。通过强化详细配置和规范，类似 Rails 的基于“约定优于配置”（convention over configuration）的原则，特别在大规模的软件生产实践中，这些约定非常重要，从无状态共享到水平扩展的过程，从松耦合架构关系到部署环境。基于 12 要素的上下文关联，软件生产就变成了一个个单一的部署单元；多个联合部署的单元组成一个应用，多个应用之间的关系就可以组成一个复杂的分布式系统应用。

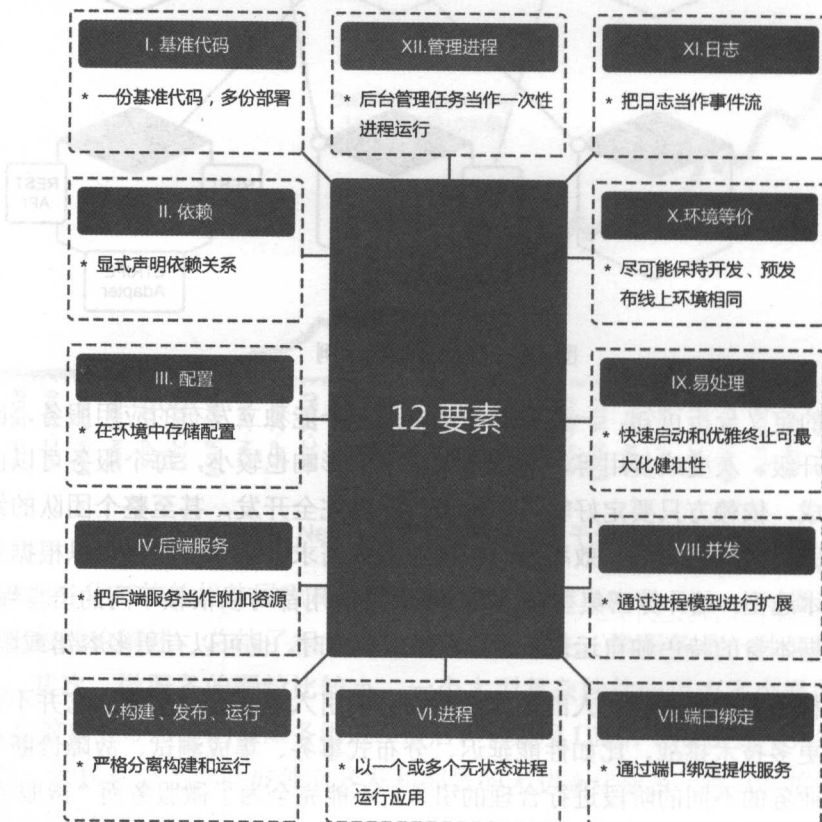


图 1-9 12 要素

下面简要介绍图 1-9 中的这些原则。相信很多开发者在实际开发工作中已经很好地应用了其中的一些原则，只是没有意识到概念本身。对这些原则比较陌生的开发者，如果想了解更多的操作过程，请在互联网上搜索《云原生时代下的 12 要素（12-Factor）应用与实践》一文作为参考。

● 基准代码

每一个部署的应用都在版本控制代码库中被追踪。在多个部署环境中，会有多种部署实例，单个应用只有一份代码库，多份部署相当于运行了该应用的多个实例，比如开发环境一个实例，测试环境、生产环境都有一个实例。

实际上，在云计算架构中，所有的基础设施都是代码配置，即 Infrastructure as Code (IaC)，整个应用通过配置文件就可以编排出来，而不再需要手工的干预，做到基础服务也是可以追踪的。

● 依赖

应用程序不会隐式依赖系统级的类库，通过依赖清单声明所有依赖项，通过依赖隔离工具确保程序不会存在调用系统，但清单中未声明依赖项，清单统一应用到生产和开发环境。比如通过合适的工具（例如 Maven、Bundler、NPM），应用可以很清晰地对部署环境公开和隔绝依赖性，而不是模糊地对部署环境产生依赖性。

在容器应用中，所有应用的依赖和安装都是通过 DockerFile 来完成声明的，通过配置能明确把依赖关系，包括版本都明确地图形化展示出来，不存在黑盒。

● 配置

环境变量是一种清楚、容易理解和标准化的配置方法，将应用的配置存储于环境变量中，保证配置排除在代码之外，或者其他可能在部署环境（例如研发、展示、生产）之间区别的任何代码，可以通过操作系统级的环境变量来注入。

实例根据不同的环境配置运行在不同的环境中，此外，实现配置即代码，在云环境中，无论是统一的配置中心还是分布式的配置中心都有好的实践方式，比如 Docker 的环境变量使用。

● 后端服务

不用区别对待本地或第三方服务，统一把依赖的后端作为一种服务来对待，例如数据库或者消息代理，作为附加资源，同等地在各种环境中被消耗。比如在云架构的基础服务中，计算、网络、存储资源都可以看作是一种服务去对待使用即可，不用区分是远程还是本地的。

● 构建、发布、运行

应用严格区分构建、发布、运行这 3 个阶段。3 个阶段是严格分开的，一个阶段对应做一件事情，每个阶段有很明确的实现功能。云原生应用的构建流程可以把发布配置挪到开发阶段，包括实际的代码构建和运行应用所需的生产环境配置。在云原生应用中，基于容器的 Build-Ship-Run 和这 3 个阶段完全吻合，也是 Docker 对本原则的最佳实践。

● 进程

进程必须无状态且无共享，即云应用以一个或多个无状态不共享的程序运行。任何必要状态都被服务化到后端服务中（缓存、对象存储等）。

所有的应用在设计时就认为随时随地会失败，面向失败而设计，因此进程可能会被随时拉起或消失，特别是在弹性扩容的阶段。

● 端口绑定

不依赖于任何网络服务器就可以创建一个面向网络的服务，每个应用的功能都很齐全，通过端口绑定对外提供所有服务，比如 Web 应用通过端口绑定（Port binding）来提供服务，并监听发送至该端口的请求（包括 HTTP）。

在容器应用中，应用统一通过暴露端口来服务，尽量避免通过本地文件或进程来通信，每种服务通过服务发现而服务。

● 并发

进程可以看作一等公民，并发性即可以依靠水平扩展应用程序来实现，通过进程模型进行扩展，并且具备无共享、水平分区的特性。

在互联网的服务中，业务的爆发性随时可能发生，因此不太可能通过硬件扩容来随时

提供扩容服务，需要依赖横向扩展能力进行扩容。

● 易处理

所有应用的架构设计都需要支持能随时销毁的特点，和状态的无关性保持一致，允许系统快速弹性扩展、改变部署及故障恢复等。

在云环境中，由于业务的高低峰值经常需要能实现快速灵活、弹性的伸缩应用，以及不可控的硬件因素等，应用可能随时会发生故障，因此应用在架构设计上需要尽可能无状态，应用能随时随地拉起，也能随时随地销毁，同时保证进程最小启动时间和架构的可弃性，也可以提供更敏捷的发布及扩展过程。

● 环境等价

必须缩小本地与线上差异，确保环境的一致性，保持研发、测试和生产环境尽可能相似，这样可以提供应用的持续交付和部署服务。

在容器化应用中，通过文件构建的环境运行能做到版本化，因此保证各个不同环境的差异性，同时还能大大减少环境不同带来的排错等成本沟通问题。

● 日志

每一个运行的进程都会直接标准输出（`stdout`）和错误输出（`stderr`）事件流，还可以将日志当作事件流作为数据源，通过集中服务，执行环境收集、聚合、索引和分析这些事件。

日志是系统运行状态的部分体现，无论在系统诊断、业务跟踪还是后续大数据服务的必要条件中，`Docker` 提供标准的日志服务，用户可以根据需求做自定义的插件开发来处理日志。

● 管理进程

管理或维护应用的运行状态是软件维护的基础部分，比如数据库迁移、健康检查、安全巡检等，在与应用长期运行的程序相同环境中，作为一次性程序运行。

在应用架构模式中，比如 `Kubernetes` 里面的 `Pod` 资源或者 `docker exec`，可以随着其他的应用程序一起发布或在出现异常诊断时能通过相关的程序去管理其状态。

云原生的内容非常广泛，目前没有系统的说明和完整的定义，上文介绍了云原生应用的基础组件和相关特点，可能读者对云原生应用的逻辑还存在一些困惑。为了更清楚地进行说明，我们总结了其依赖关系，如图 1-10 所示。

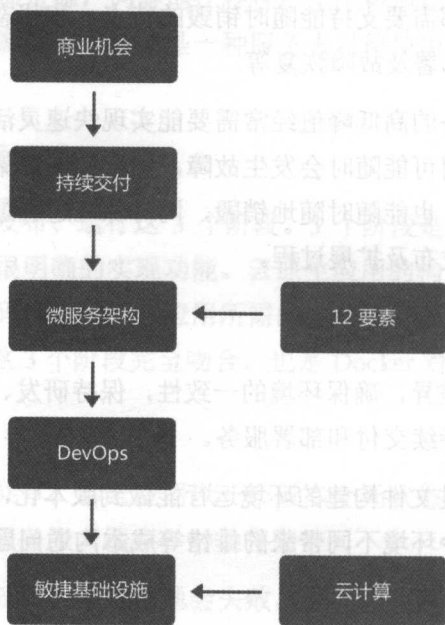


图 1-10 云原生内容的依赖关系

首先，为了抓住商业机会，业务需要快速迭代，不断试错，因此，企业需要依赖拥有持续交付的能力，这些不仅包括技术需求还包括产品的需求，如何能拥有持续交付的能力，大而全的架构因为效率低下，显然是不合适的。于是演变出微服务架构来满足需求，通过把系统划分出一个个独立的个体，每个个体服务的设计依赖需要通过 12 要素的原则来规范完成。同样，如果系统被分成了几十个甚至几百个服务组件，则需要借助 DevOps 才能很好地满足业务协作和发布等流程。最后，DevOps 的有效实施需要依赖一定的土壤，即敏捷的基础设施服务，现实只有云计算的模式才能满足整体要求。通过上述梳理，我们总结出面向云原生应用的 3 个不同层次的特点。

- 高可用设计 (Design for Availability)，依据应用业务需求，高可用分为不同级别，比如不同区域、不同机房（跨城或同城）、不同机柜、不同服务器和不同进程的高可用，云原生应用应该根据业务的可用性要求设计不同级别的架构支持。

- 可扩展设计 (Design for Scale)，所有应用的设计是无状态的，使得业务天生具有扩展性，在业务流量高峰和低峰时期，依赖云的特性自动弹性扩容，满足业务需求。
- 快速失败设计 (Design for Failure)，即包括系统间依赖的调用随时可能会失败，也包括硬件基础设施服务随时可能宕机，还有后端有状态服务的系统能力可能有瓶颈，总之在发生异常时能够快速失败，然后快速恢复，以保证业务永远在线，不能让业务半死不活地僵持着。

通过上面的基本描述及云原生应用的组成或特点，与容器技术（第 2 章将详细介绍）相比可以得知，容器的特性天生就是按这些原则进行设计的。随着互联网业务的架构不断演进，从单体应用到分布式应用，甚至微服务架构应用中，12 要素较好地构建互联网化应用提供了统一的方法论和标准化，具有强大的生命力，每一条原则都是应用开发的珠玑。当然，在实践过程中，每一个原则也不是一成不变的，随着新的理念和技术出现，原有的因素会得到延伸和发展，会出现新的原则和应用，这套理论也适用于任意语言和后端服务（数据库、消息队列、缓存等）开发的应用程序，因此也作为云原生架构应用的基本指导原则之一。

1.4 架构演化发展历程

目前，互联网企业随着业务的发展不断前进。因此，不同的阶段有不同的需求，所以需要使用不同的方法来聚焦不同的目的。比如初创型的企业需要抓住合适的机遇快速进行原型验证，证明大的方向没问题才有可能进一步发展。因此技术一般是为了满足业务的发展和验证而设计，只要保证共用的组件能够复用就可以。只有随着业务的进一步发展，组织架构及人员的规模越来越大，企业才有机会发展成中型或独角兽公司。然而协作人员数量的增长也意味着新的问题，一般而言，只要企业的人员规模超过 150 人，就会带来极大的管理成本提升。在这一阶段，技术不仅需要和产品进行更充分的结合，还需要规范化和工程化操作。当规模再次扩大的时候，如果只是简单的人力堆积，协调性问题是无法解决的，企业需要更高效的架构才能支撑和反哺业务的发展，工具和流程需要更加自动化才有可能保证业务的持续发展，边际效应非常明显，同时挑战也更大。

总之，在任何阶段，技术的演进都是一个非常重要的过程，这一点在以技术为导向的公司尤为凸显。技术超前就有可能成为先行者，技术落后就有可能导致企业发展缓慢，机

会丢失。

无论是个人开发者还是技术公司，如果目的是持续运行和发展，就需要一套良好并固定的制度，同样，在技术上，团队的持续前进离不开项目的工程化思维。开发、测试、发布的软件生命周期的高效管理必须依赖工程化，它是研发协作与云端运维的基础。这里面包括许多问题，需要技术人员去解决，比如：追求技术的先进或业务需求的合理性评估，可能是很多公司技术管理需要根据组织架构权衡的问题。一般来讲，任何从零起步的公司都会经过不同的技术发展阶段，包括简单的初创期、快速增长期及分布式的服务化架构阶段。每个阶段对应不同的产品和业务需求，技术承担不同职责，需要技术人员来决策，尤其是架构师需要思考这些问题。

1.4.1 初创期架构

创业公司在开始新业务的时期，基本处在试错或原型验证阶段，这个阶段更多是关注业务的本身是否有前景或商业模式，而不会把非常多的精力放在技术的系统架构上，尤其是对于非技术型或不确定型的项目立项阶段。尽管很多技术人员也预料到前期需要很多时间去好好设计系统，才能保证支撑后续可能的业务快速发展，但往往由于时间成本或人力等原因而无法很好地执行。

一般来讲，创业型的项目对时间的要求非常苛刻，需要在 3 到 6 个月时间内完成系统的上线，否则有可能由于业务无法快速上线验证，导致无法获取相关的原始数据进行下一个目标验证，更严重的有可能造成资金链的断裂。罗马不是一天建成的，因此这个阶段会使用相对简单的架构方式来进行设计，本节先从最主要的几点进行说明，更详细内容请参考第 3 章。

单体架构

对于创业型公司来说，由于人才、技术、资金等重要因素的影响，同时，技术人员为了配合产品的需求，会采用最简单的架构来完成最原始阶段开发。根据我们接触的不少用户反馈，有些企业考虑成本因素，甚至只使用一台服务器或者容器服务。另外，传统官网、论坛等应用，由于早期的设计采用了单体架构来实现，只需要一台服务器或容器来服务即可。对于其他的应用服务器、数据库、静态文件等资源，也是部署到同一台服务器或容器上来服务。最简单的架构模型如图 1-11 所示。

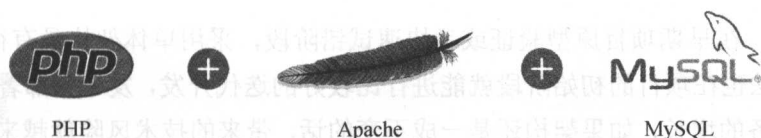


图 1-11 最简单的架构模型

对于早期的单体应用，应用服务+数据库服务基本上就组成了最原始的架构模型，技术人员更多会考虑技术的选型，包括编程语言、版本管理、数据库的类型等。比如 PHP 的开发者选择 PHP+MySQL，Java 的开发者采用 Tomcat+MySQL 等开发方式。

服务器分离

根据线上运行经验，一般的业务的类型，如果每日的用户访问量在百万级别以内，只要进行简单的 Web 应用性能参数调优、数据库索引优化等，基本上就能保证服务的稳定运行。当然，随着访问量的不断增加，部署在同一台服务器上的应用及数据库服务，会造成服务器的 CPU/内存/磁盘/带宽等系统资源竞争，从而相互影响。显然很容易出现性能瓶颈，如果这台服务器出现了宕机或无法恢复的错误，就有可能导致全站不可访问或者数据丢失等情况，后果非常严重，因此大部分产品会将 Web 应用服务器和数据库服务器进行物理分离，独立部署，相互热备提供服务，只需要增加很少的成本，就能解决对应性能和数据的可靠性等问题。

初期由于各种条件存在，不能很好地进行新项目前景的预见，技术人员如果能在最小成本的情况下保证架构的合理性，还能很好地服务产品功能需求，甚至只要在部署架构上稍做调整，就可以防止出现灾难性的问题，这其中也包括很多技术架构上的考虑。

业务模型

一般而言，现阶段的业务比较简单，产品也比较单一，业务会随时根据其运营数据进行调整，因此，这时需要技术人员能够较好地把不同的模块分离出来，对于偏业务相关的功能，需要有较好的心态接收随时变化的不确定性，对于后续可能复用或大量依赖的工程，需要进行较好的设计，否则可能在业务爆发时导致业务开发的进度越来越慢，甚至阻碍业务的发展，造成业务时常中断。即使有人力或时间来对系统进行重新设计，也会令技术人员产生抵触心理，同时也会引入较高的风险。因此，基于云原生应用的设计模式在最基础的阶段对架构也有很大的作用，包括考虑如何使用云的弹性，将不可变优势融合到系统的设计中。合理的业务模型分界也是确保后续能发展的重要步骤之一。

总而言之，在早期项目原型验证或者快速试错阶段，采用单体架构具有很大的技术优势，产品的想法也在项目的初始阶段就能进行比较好的迭代开发，发布和部署也比较灵活。然而，随着业务的增长，如果架构还是一成不变的话，带来的技术风险就越来越高，比如，代码行数的增加影响技术人员的学习成本、业务的变更速度、业务的可靠性、安全性及工程变大后的发布效率等，每次修改都必须反复测试，否则全站随时可能不可用，导致业务中断或者丢失市场的机会。因此，这部分的技术债务必须在业务快速发展的同时，进行技术架构的改造，使其能保证后期业务的支撑。所以，除了业务的发展判断外，对开发人员的技术能力储备和架构远见判断也将成为考虑的事情之一。

1.4.2 快速成长期架构

接下来，初创公司随着业务的进一步发展，当 DAU 达到十万的时候，通常是最关键的时刻，既要保证业务的稳定运行，又要进行产品的快速迭代。到了这个阶段，由于业务模式得到了一定的验证和反馈，有可能会出现很多竞品或友商。一方面，随着风险资本的注入，会依赖更有质量的数据进行发展运营，另一方面，竞品的出现又导致了市场的加速前进。因此，能否在这个阶段保证业务与技术的和谐发展，是考验架构是否足够灵活的指标之一，同样，本节主要说明几点，更详细内容请参考第 4 章。

前端加速优化

首先基于浏览器端应用或者移动端应用，随着请求的不断增加，偶尔会看到 Web 服务出现性能瓶颈导致请求变慢或者失败。除去服务器本身的配置低外，更有可能由于架构设计或分离的原因，大量的 Web 并发请求被堵塞或者变慢，原因无非是服务器的 CPU、磁盘 I/O、带宽竞争激烈，导致相互影响。这时候我们就需要对架构进行前后端分解，合理配置或转发请求，如果是前端的服务请求来不及处理或者有瓶颈，可以将图片、JS、CSS、HTML 及应用服务相关的静态资源文件存储通过 Nginx 本地代理或者对象存储服务来进行物理加速，使用不同的域名来转发请求，并通过 CDN 将静态资源分布式缓存在各个节点实现“就近访问”，主动或被动刷新 CDN 的缓存来加速前端服务。如果是后端的动态请求压力过大或者有热点服务，可以把无状态的后端的服务再进一步水平扩展满足业务分担，有状态需要判断是否能够通过垂直扩容来服务，否则只能进行代码、架构设计或者业务规划的调整来优化。一般而言，通过将动态请求、静态请求的访问分离（“动静分离”），能有效解决服务器在 CPU、磁盘 I/O、带宽方面的访问压力，当然，这需要在架构设计时采用一些方法来

进行调整。

水平扩展

在上节中提到垂直扩容能解决部分的问题，但由于业务和流量的快速增长及垂直资源有限，不同的应用场景需要依赖不同的策略分流，比如长连接的应用会依赖于 4 层的网络连接，互联网应用通常采用 7 层的模式来完成，甚至在游戏场景中，依赖 UDP 进行通信。为了更多地分担服务器的压力和保证业务的高可用，负载均衡技术通常是这个阶段解决问题的一个方法，通过增加多台后端服务器就可以实现分流的功能，分流设计也面临很多原则与技巧，比如分流的路径、权重等，负载均衡承担的角色也决定了后端的应用架构，比如无状态化设计才能实现水平扩展，另外还要考虑业务是否有亲缘性，同时在后端服务出现异常的情况下，自动进行健康检查，异常的服务能及时进行下线操作，快速失败。

数据库及缓存优化

数据库和缓存配合使用是解决后端结构化数据与非结构化问题的有效手段，根据不同的场景，要明白哪种数据使用结构化数据合适，哪种数据使用非结构化数据更合适，以及哪种方式在保证性能较好的情况下成本又可以接受。同时，如何在数据库和缓存之间进行过渡也是需要考虑的，比如数据在更新的时候，如何保证缓存的一致性，如何保证热点数据一直被访问，提高缓存的命中率等。另外，当大量用户访问不存在的数据时，也有可能导致后端的压力非常大，甚至有可能造成雪崩效应。

每种服务独立承担对应的功能，各司其职，并且根据应用的特性区别提供不同的服务能力，比如应用服务器提供用户的接入服务，数据库服务专门承担结构化数据的存储，缓存承担或非结构化数据（KV 值对）的存储等，如果要提供搜索的功能，还需将数据进行分词、索引、检索等，不同服务器根据业务的功用需求来提供对应的服务，如图 1-12 所示。

在这个阶段，除了必须保证满足业务的功能型需求，还要更多考虑非功能性需求，比如，通过前端负载均衡提供业务分流的能力，根据用户的特征进行不同的流量转发；数据库提供主备的能力，两者之间通过数据同步进行数据备份，当主数据库发生故障后，应用可以自动切换到备份的服务器来为用户提供服务；在用户体验方面，可能会引入缓存、CDN 等基础服务来提供性能加速。

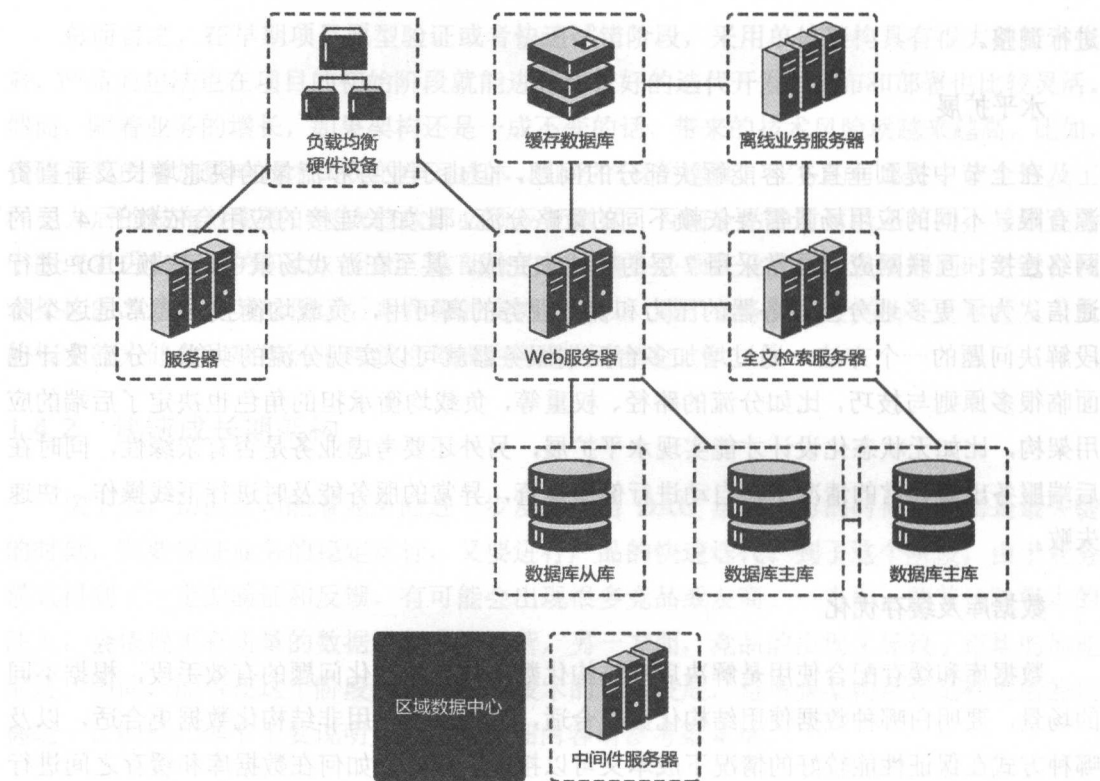


图 1-12 数据库及缓存优化

1.4.3 分布式服务架构

一般而言，企业经过前面两个阶段的发展，就基本确定了业务的发展方向，接下来只要面对竞争对手的跟随和大量用户访问请求的问题。这些企业也会提供各种不同的子产品模块功能来满足业务的多样性发展，比如产品会设计不同的产品功能体系，运营人员会设计不同的运营活动，客服人员会接到不同的用户反馈等。这些需求叠加在一起，会导致整个业务越来越复杂，有些系统变得不再具有维护性，无法满足可用性的需求，只要一出现流量高峰，系统必定宕机，所以很多公司面临重构架构设计，甚至推翻重来，比如京东大部分的用户业务从.NET 转到 Java 语言的重构，淘宝从 PHP 转到 Java，升级 2.0，再到 3.0 的架构转变，可以看到，这个时间点的转型或重构可能不会到生死攸关的地步，但其成本是非常高的。因此，如何在一开始就能做好这种转变的预见性，对架构设计有相当大的挑战，本节先给出几点供读者来参考，更详细内容请参考第 5 章。

业务需求

通过云服务通常可以解决很多架构层面的问题，比如对象存储系统解决了文件分布式存储的问题，CDN 解决了静态资源访问的性能问题，但实际上随着业务的不断发展，系统访问压力增大，还可能有很多请求变慢或者超时，应用服务器或数据库服务的压力波动较大，只要不停地上线新业务，技术债就会越来越明显，业务的迭代也越来越跟不上产品需求。为了解决这些问题，企业往往需要进行各种业务的拆分，把不同的功能模块拆分到不同的服务器上独立部署。比如用户模块、商品模块、购物车模块、订单模块和支付模块等，这些模块拆分并独立部署出来后，可以更进一步根据系统的瓶颈进行细分。但是进行服务拆分之后，各模块之间的依赖又变得明显起来，比如数据库的连接数、数据的分布式事务、数据库的性能开销等都是急切需要解决的问题。

同时，随着业务模块的拆分，除了上述的技术问题要解决外，还面临着工程实践的问题，比如在业务的不同分支中，需要保证开发人员、测试人员、运维人员快速地开发环境、测试环境、预发布环境的搭建和发布。在高速发展的企业中，迭代的频率非常高，以网易考拉平台为例，所有系统的日发布总次数到达数千次，所以技术人员对效率的要求极高。当扩大到一个公司多个产品线，整体的运行就要求像现代化工厂一样来运作，需要自动化的能力平台去解决，纯手工根本无法满足企业的高效运转。

弹性扩容

随着需求和用户的不断增长，系统会出现波峰和波谷，为了更好地利用资源和成本预算，弹性扩容成了必要需求，在峰值的时候能够根据业务的压力自动扩容，分担流量，在压力低的时候自动缩容，减少成本或提高资源的利用率，把缩容的资源做离线业务计算。也许在过去是简单地通过垂直扩大规模能力来处理更多的需求，或者是购买更强的服务器，这在一定程度上是可行的，但过程很慢并且代价庞大，通过提前准备过多的资源，会导致只根据峰值使用量预测来规划能力值，比如根据服务器的最高计算能力购买硬件予以满足，这是不得已的做法。例如国外的黑色星期五、国内的双 11 等活动，当天请求非常高，需要足够的资源来满足业务请求，而平时这些服务器的使用率很低，所以，只有依赖云的弹性才能满足这种业务场景的需求。

服务化

不管是互联网还是传统行业转型的企业，基本都是在原有基础业务上发展而来的，不

可能把业务停止从零开始，因此，直接对原有系统进行微服务改造比较困难，风险也较大。这时基本上处于一种混合架构期，即新的业务会从头开发，逐步接入到老系统中，一步步替换老系统不满足的地方，通过不断地快速迭代来保证业务的可持续性，同时又保证新业务的快速需求。著名的架构大师 Martin Fowler 从 2013 年正式提出微服务架构的综述文章至今，都没有提供统一的最佳实践方案。现在微服务的架构实现方式也各种各样，通常根据应用的类型拆分成不同的微服务来实现，每个服务根据业务的特征采用不同的技术栈进行组合，把每个服务划分成可以独立部署的隔离进程来运行。目前，微服务的基本框架都类似，比如包括服务发现、降级、治理等方面。业务实现微服务的技术细节各不相同，没有统一的实现方案，比如服务发现有自建服务基础设施的，也有依赖第三方开源的，技术人员需要根据自己的场景来做选择。简化的架构模型如图 1-13 所示。

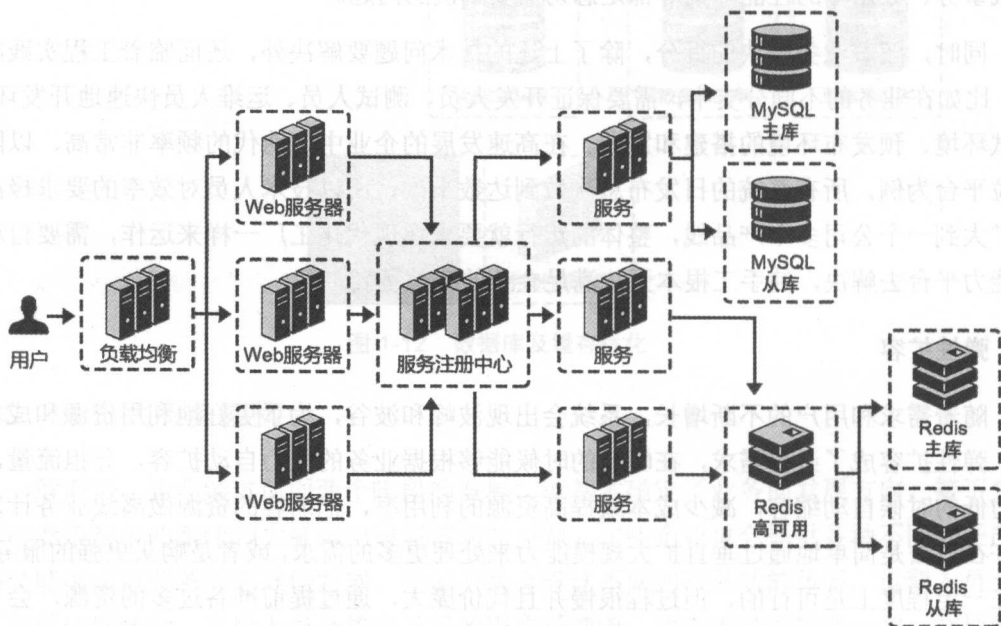


图 1-13 服务化架构模型

显然，这个架构模型只是整个业务服务架构的一部分，实际的系统可能要复杂几十倍。如果业务的迭代速度非常快，同时每个业务之间的依赖从设计、开发、测试、上线到运维都是一个非常庞大的复杂工程，因此，如何高效管理所依赖的服务和系统依赖，诊断并及时对业务反馈响应是对服务架构的考验。

1.5 云计算服务介绍

目前，早期的云服务厂商，通过资源提供的方式，将云计算划分为 IaaS、PaaS、SaaS 3 层服务模式，每层提供不同服务的使用范围和功能来满足不同的使用场景，并且这 3 层有明显的层次关系：最上层的 SaaS 将应用作为服务提供给客户，无须开发者有软件背景，直接使用软件就可以提供服务，比如 CRM、HR 等系统；第二层的 PaaS 将一个开发平台作为服务提供给用户，一般需要开发者有相关的技术背景，开发者通过自行开发相关的代码来使用平台的服务，最终为用户提供服务；最底层的 IaaS 将虚拟机或者其他资源作为服务提供给用户，一般要求使用者有较强的技术能力才能比较好地掌握和利用好资源，只提供最基本的计算、网络和存储等能力，同时也提供了更多的灵活性。

后来随着用户的需求越来越多，云服务厂商加入了越来越多的产品来为用户服务满足业务的不同架构，从 IT 架构、数据架构到应用架构。比如，目前的云服务厂商，有国外的 AWS、Azure、Google 和国内的阿里云、腾讯云、网易云等。鉴于云服务的通用性和独特性，下面以网易云为参考示例对云基础服务进行说明，其他云计算厂商的服务，请读者自行参阅相关资料。

网易云基础服务是网易公司推出的场景化的虚拟机云和容器云平台，深度整合了 IaaS、PaaS 及容器技术，提供弹性计算、DevOps 工具链及微服务基础设施等服务，帮助企业解决 IT、架构及运维等效率问题，使企业更聚焦于业务，以改变互联网软件的生产方式。目前主要提供 3 类的服务来迭代目标：第一，云计算基础服务（计算、网络、存储等）；第二，融合稳定的平台服务（关系数据库、负载均衡、缓存、对象存储等）；第三，提供丰富完善的 DevOps 工具链和多样的微服务基础支持。

服务端架构的设计是大部分技术人员在业务设计之初就会考虑的问题，技术人员既希望能支撑业务的快速迭代，又希望能采用优雅高大上的架构，比如一开始就采用 SOA、微服务或分布式架构。实际上，要较好地应用这些服务的架构是很有挑战性的，既要搭建微服务基础设施，同时还要对服务进行编排，最后进行服务的治理、升级、监控等。为了实现微服务的快速交付，在日常的开发中，团队需要进行开发测试线上环境的快速搭建，保证环境一致、持续集成、一键发布等过程，这些过程大部分是烦琐或者低效的，采用 DevOps 的协作方式能比较好地解决这些效率问题，提供的容器服务、镜像服务和持续集成等基础服务就是为了实现 DevOps 而设计的。

云基础服务具体包括容器服务、主机管理、镜像仓库等产品，这些基础产品提供基础的计算、网络和存储等能力，另外，对网络隔离有要求或者规划较好的企业还要依赖虚拟网络（VPC）等服务。容器和主机提供高可用、高性能、弹性伸缩的计算、存储和网络服务，以达到在线扩容、快速响应业务变化，提升交付效率的目的。产品的性能和实现敬请关注官网的参考数据和文档，在此不再详述。总之，这些服务能高效支撑各种互联网使用场景，共同构建提供企业级闭环解决方案的基础服务，以网易云为例，整体的产品基础服务架构如图 1-14 所示。

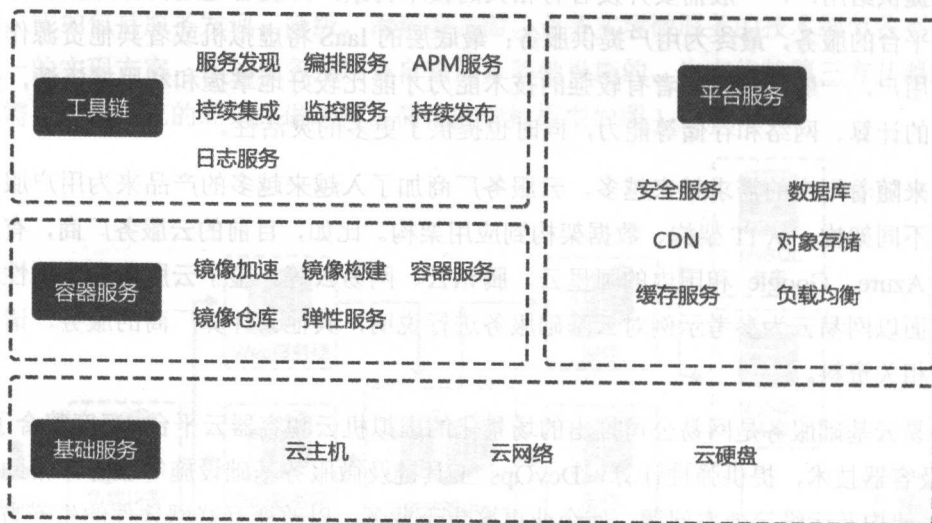


图 1-14 网易云基础服务架构

在平台服务层面，网易云提供负载均衡、数据库（RDS）、缓存服务（Redis）、对象存储（NOS）、CDN、安全服务和 IP 管理等平台级服务。高可用的负载均衡配合多规格的容器组成的微服务架构可以很好地支撑用户的弹性需求。关系型数据库和 Redis 缓存可以支撑各种常见的访问需求和数据存储能力，搭配对象存储的图片上传处理和 CDN 加速等技术，大大减少架构的复杂性，可以高效地弹性扩展，适应负载高峰。在安全层面，DDoS 安全、账号认证和安全验证等多重限制可以极大程度地保障用户的数据免受外界冲击。

此外，由于国内网络环境的特殊性，还会提供相关的业务运营支撑系统，比如域名和备案系统，可以完全满足企业域名的购买、域名的解析，以及对应的备案；弹性 IP 将允许用户保留自己的 IP，方便后续的备案、变更及应急切换。

总之，网易云提供丰富、稳定和可靠的基于场景化的云服务，云基础服务是网易云产品体系里面最基础服务，容器服务又是云基础服务的最核心功能。此外网易云还有一大特色，就是依赖产品体系、知识体系和服务体系来保障业务与云技术水乳交融。本书也是知识体系的重要组成部分，除此之外还有通信、安全等产品体系，以及产品、前端开发和大数据等知识体系。

1.6 云计算解决方案

在云计算及服务中，我们会针对某个具体已经体现出的或者可以预期的问题、不足、缺陷、需求等提出一个解决整体问题的方案，以及制定确保能有效执行的计划表。由于问题的多样及复杂性，常常会涉及很多系统及人的交互场景，因此产生不同的解决方案来处理，需要在不同的阶段选择不同的方案来满足需求。

正如乐高积木提供的 1300 种不同形态的凸凹槽的积木，包括每个积木采用 12 种不同的颜色，用户根据自己的喜好采用这些基础的组件就可以拼装组成一座座美妙的三维图形，从最简单的积木到复杂的城堡、庄园等，被很多用户所喜好。云计算的基础设施提供的就是这样一种能力，服务既可以提供简单的基础组件，又可以提供封装好的组件，开发者根据业务的需求就能组成一系列的产品来服务于用户。

由于行业的多样性，云计算对行业的分类也非常广泛，按照相关的统计与行业的统称，大致分为电商、金融、O2O、教育、视频、社交、游戏等行业。实际上每个行业都有不同的解决方案，即使是同一个行业，也会采用不同的方案来完成相关的业务，不可能有完全一样的方案，因此网易云也只提供多种不同场景的解决方案供读者参考，相信开发者都有能力根据业务的场景要求独立完成自身行业方案的建设。

大部分的云计算服务厂商也提供了各行各业的解决方案，尽管方案会有一些不同，但是核心的内容相差不多，所以，下面仅以网易云为网易内部的电商平台提供的案例来说明其应用，供开发者参考。

如图 1-15 所示，网易云电商行业解决方案整个过程如下。

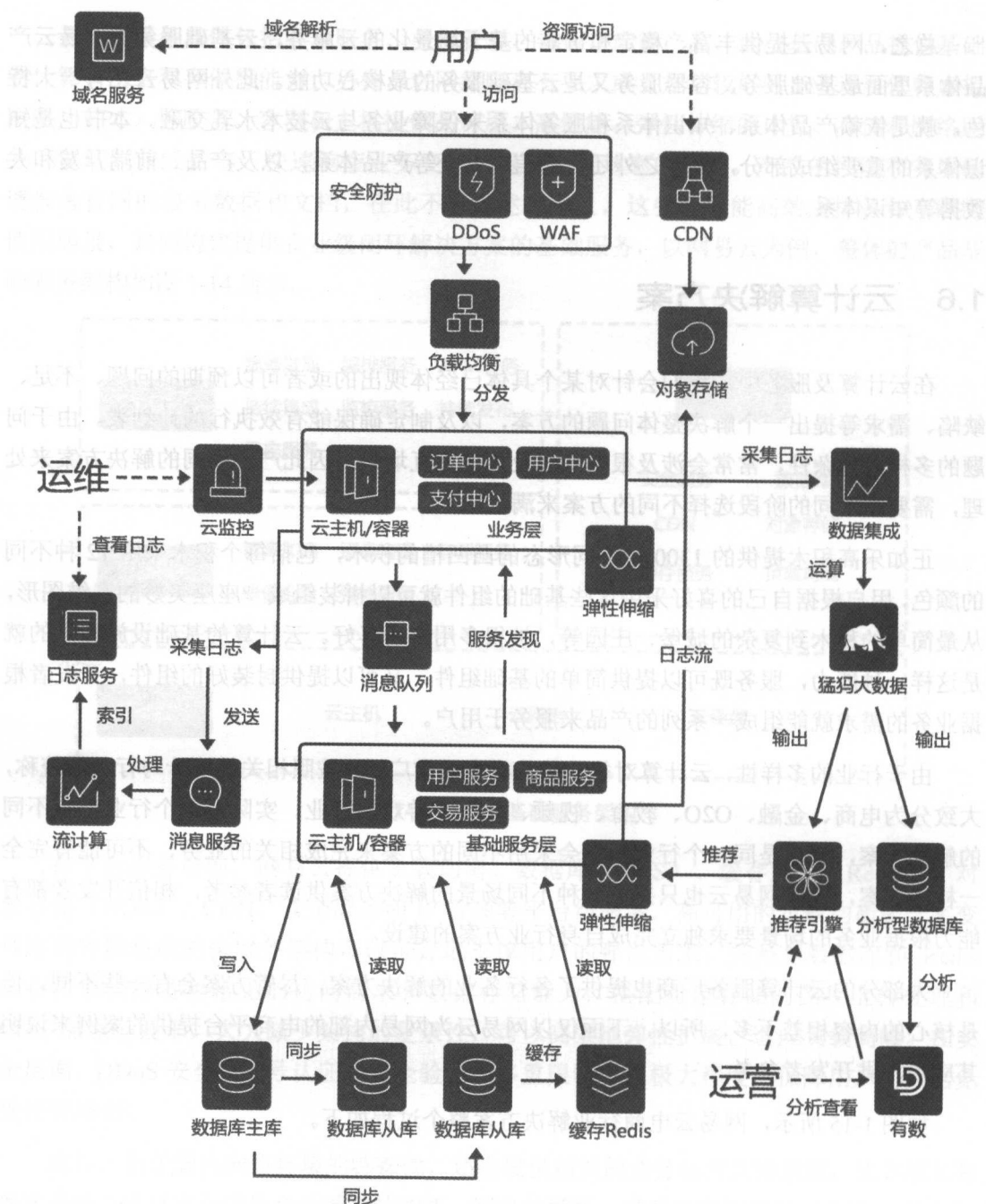


图 1-15 网易云电商行业解决方案

1. 用户通过云厂商注册的域名访问 DNS，域名服务器将域名解析为 IP 地址，浏览器根据 IP 请求服务。

2. 当访问静态 Web 页面时，直接从 CDN 读取，如果 CDN 不存在资源，则回到对象存储服务上获取，并在 CDN 服务上缓存。

3. 当访问动态页面时，首先通过安全防护模块和应用防火墙，如果正常就把请求转发到后端服务，否则拒绝请求。

4. 负载均衡根据请求和规则将请求分发到后端服务，先访问的是业务层的 API 服务。基础通用服务一般提前在服务注册中心注册，API 业务通过服务发现调用各种服务组合业务，异步业务层通过消息队列或者异步调用基础服务。

5. 基础通用服务层写入主数据库，主数据库同步到多个从数据库，为了加速基础服务访问速度，减少性能开销，直接从 Redis 缓存读取热点即可。

6. 所有服务的业务日志可通过日志流将日志发给消息服务 Kafka。

7. 流计算从消息服务 Kafka 读取数据进行处理，将结果写入日志服务提供开放搜索。

8. 日志文件从消息服务 Kafka 读取数据上传到对象存储帮助交易数据进行大数据处理，通过猛犸大数据平台（EMR）进行大数据计算，计算的结果部分进入分析型数据库，部分进入推荐引擎。

9. 运营人员读取分析型数据库的数据，进行敏捷的数据分析或可视化操作，帮助用户进行业务决策。

10. 产品人员可以利用推荐数据来更新用户的视图，推送给用户，进行下一轮的迭代。

上面是网易电商平台线上完整运行的一个案例，当然，这只是企业业务上线后的最后架构运行状态的一个结果，实际上，这个结果的有效运行是依赖企业组织人员一起努力的工作，是整个互联网软件生产中的一小部分，前期还包括架构设计、开发、测试、运维优化等环节，比如，出问题时搜索日志查看问题的原因，通过云监控查看监控信息及设置服务能支持弹性扩容的条件。事实上，即使初创产品有机会上线，产品也需要不断迭代更新和发展，否则很多业务无法走到这一步或者走的过程非常漫长。软件生命周期在产品里程碑中占用的时间非常多，两者相互制约，相互促进，如果架构设计不是很优雅，当后续的业务爆发增长时，就可能导致大量的重构和技术债，所以在设计之初就能考虑周全，受

益是非常大的。

1.7 案例概述

无论是互联网行业还是传统行业加入互联网，项目的展现形式都各不相同，前文提及根据行业的类型，传统行业分为信息、金融、教育、食品、建筑、制造、批发和零售等领域，种类丰富。越来越多的传统行业积极加入互联网的浪潮中，进入到采用互联网思维进行企业的变革和转型中，所以，无论如何大部分企业都需要转型，采用云服务的思维加速企业变革。

变革是一个漫长而又痛苦的过程，除了思想的改变之外，还包括很多方面，比如组织机构、行政架构、技术架构甚至业务方向等。在这里，我们更多是探讨技术架构的上“云”之路，如何根据自己的业务发展特点，在不同的阶段采用不同技术特征适合业务的发展。

1.7.1 背景介绍

互联网业务丰富多样，根据业务的不同，我们通常会在不同阶段使用不同的服务架构来支撑业务的发展，因此，为了让读者能够动手实践所有内容，我们根据各行业应用架构发展的一些共同特性，提供单体架构、分布式架构和微服务化架构 3 个不同阶段的示例为源代码给读者提供代码级别的参考，以麻雀虽小五脏俱全的案例为背景，提供完整的案例供读者练习，来演示架构的变迁过程。

此外，为了给读者提供更丰富的内容和保证解决方案示例的一致性，我们选取了电商架构细节的内容作为本书的首选案例，对场景进行详细说明，原因之一是电子商务类的需求也是大部分互联网及互联网+应用的诉求，比如电子商务通常包括会员、订单、购物车、秒杀、详情页、评论、支付、物流等模块，后台还会包括各种管理系统，比如库存管理、发货管理、优惠券等模块。此外，购物类网站包括的技术种类非常多，而且大规模的爆发场景、支付一致性的场景等所依赖的技术非常具有代表性，无法简单地在源代码级别提供示例。

我们希望每个用户都能在接下来的各个章节中找到对应的位置，根据业务的特点进行有机组合，实现更好更快的上“云”之路，下面对源代码示例进行简单的说明和介绍实际

操作步骤，读者可自行选择阅读或者略过直接进入第 2 章。

由于源代码里面有详细的指导说明文档，所以此处也不详述，欢迎读者在 Github 给开发人员提意见，以展示更多的架构能力给其他读者，项目代码整个示例分为 3 个阶段，分别如图 1-16、图 1-17、图 1-18 所示。

1. 单体架构（应用和数据分离）

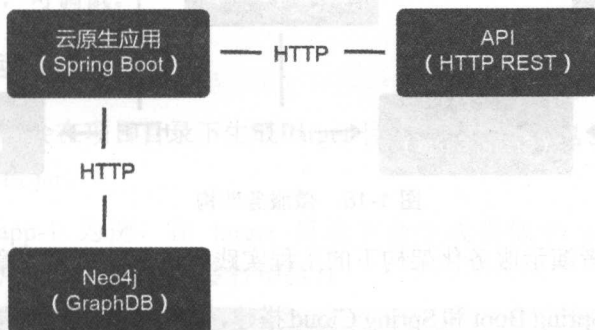


图 1-16 单体架构

本分支主要给读者演示满足最基本需求的产品原型的工程实践，是第 2 章和第 3 章的内容实践操作。

2. SOA 架构（前后端分离，服务化）

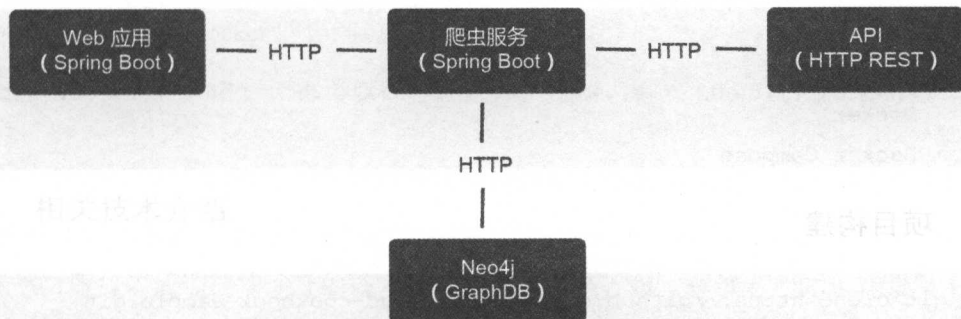


图 1-17 SOA 架构

本分支主要给读者演示满足企业在快速成长期的工程实践和挑战，是第 4 章的内容实践操作。

3. 微服务架构（服务注册与发现、分布式配置管理、负载均衡、服务网关、断路器）

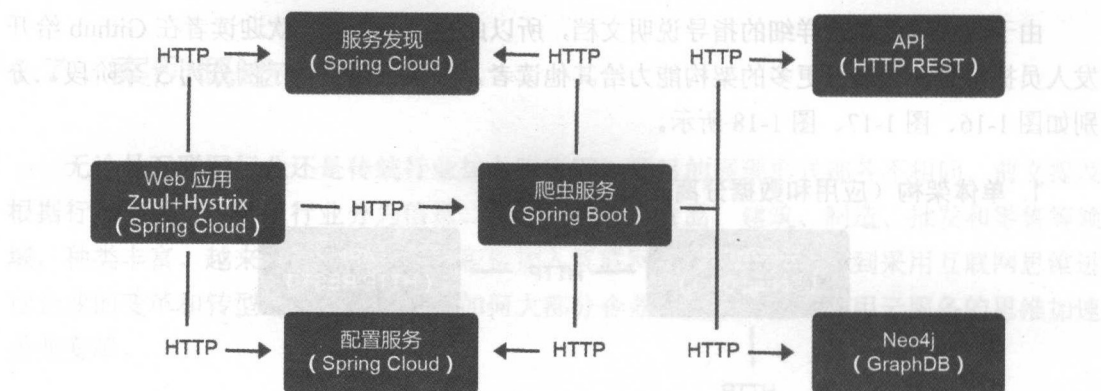


图 1-18 微服务架构

本分支主要给读者演示服务化架构下的工程实践和挑战，是第 5 章的内容实践操作。

本项目都是基于 Spring Boot 和 Spring Cloud 搭建，使用 Maven 作为构建工具，同时 Maven 构建流程中包含 Docker 镜像的构建。在构建项目前，请先配置好运行环境，接下来将介绍环境要求。

1.7.2 环境要求

本机需要安装如下环境。

- Maven 3
- Java 8
- Docker
- Docker Compose

1.7.3 项目构建

```
git clone https://github.com/163yun/cloud-cookbook-sample.git
cd cloud-cookbook-sample
mvn clean install
```

1.7.4 项目运行

项目运行的方法有多种，读者可根据自身的习惯自行选择，以下介绍常用的 3 种使用

方法供读者参考。

1. 通过 IDE 运行

使用 IDE 导入工程找到包含 `main` 方法的 Class，比如 `cloud-native-app-1` 中的 `CloudNativeApplication` 类，右击，选择 `Run` 即可。

优点：快速简单，可调试。

2. 通过命令行运行

项目构建完之后，会在项目目录下生成 `target` 目录，该目录中包含可执行的 `jar`，可以直接在命令行中启动该 `jar`。

以 `cloud-native-app-1` 为例，在 `target` 目录下会生成类似于 `cloud-native-app-0.0.9-SNAPSHOT.jar` 的可执行 `jar` 包，在命令行中执行

```
$ java -jar cloud-native-app-0.0.9-SNAPSHOT.jar
```

即可在当前控制台中看到日志输出。

3. 通过 Docker 运行

在每个项目下，都有一个 `start.sh` 脚本，该脚本可以方便地启动 `Docker` 容器，前提是需要安装好 `Docker`，详情请看每个脚本说明。读者如果暂时没有相关的知识背景，可以先跳过本段，第 2 章有 `Docker` 的详细说明，请参考后运行。

通过以上 3 种方式之一，都可以在本地打开浏览器，输入 `http://127.0.0.1:8080/` 来访问应用。

1.7.5 相关技术介绍

显然，服务端后端架构涉及的技术非常多而且复杂，每一位读者的知识背景又不太相同，所以这里不可能一一列举所有使用到的技术。但为了让读者更好地理解后续章节的内容，这里预先简单介绍本书中引入的一些关键技术，如需要更详细的信息，请读者查看附录提供的内容或网上自行搜索相关的资源来补充。

Docker

Docker 是 2013 年开源的应用容器引擎，开发者可以根据配置文件把应用及依赖包放到一个可移植的容器中，然后发布到一定版本以上的任何流行的 Linux 机器上，实现轻量级别的虚拟化。容器完全使用沙箱机制，通过镜像来保证运行环境的一致性，启动速度在秒级别之内，以更好地满足云计算的自动化及弹性扩容等场景。

使用 Docker 可以：

- 保证环境一致性，使应用的构建与部署自动化。
- 提供轻量级别的 PaaS 环境使之持续集成效率更高。
- 通过镜像服务提供完整的软件运行环境，开箱即用。
- 单个容器承担独立的职责，轻量级，启动速度快。
- 标准的镜像服务为多云部署能力提供基础条件。

Kubernetes

一个产品通常由多个应用组成，容器只是提供一个应用服务的能力，需要把多个应用组合编排起来才能提供服务。Kubernetes 是自动化编排容器应用的开源平台，这些操作不仅包括部署、调度和节点集群间扩展，还包括服务发现和配置服务等架构支持的基础能力。此外，Kubernetes 不仅支持 Docker，还支持 Rocket 等不同的底层容器技术。

使用 Kubernetes 可以：

- 提供空间（Namespace）级别环境隔离，自动化编排多个容器的部署。
- 根据应用负载压力，随时扩展或收缩容器规模及副本数。
- 将容器抽象成服务层（Service），服务间提供负载均衡和发现的能力。
- 提供滚动升级的能力，很容易地升级应用程序容器的新版本，失败时自动回滚。
- 提供 Pod 的组合多个容器，共享内核资源。

CDN

CDN（Content Delivery Network，内容分发网络）是内容网络提供商部署在各城市机房里面的一种服务。它能在用户请求网站服务时，根据用户的来源从距离自己最近的网络

提供商获取数据，减少网络路由等开销，应用在视频网站、软件下载和内容网站的热点内容的场景。

使用 CDN 可以：

- 加速网站或应用的静态内容分发，提升网站的访问速度和服务可用性。
- 减轻源站负载，提高文件下载速度，减少用户下载时间。
- 分摊流量，节省成本，特别是提升用户在不同范围内的视音频体验。

负载均衡

负载均衡是将负载（用户的请求）根据设定的策略，将请求流量进行分发，扩展应用集群对外的服务能力，提高应用的可用性。既可以提升服务器的响应速度及利用效率，又能避免出现单点失效，检查后端服务的正确性，保证用户请求的正常响应。

负载均衡的作用如下。

- 请求分流，提供 4 层和 7 层负载均衡能力。
- 支持自定义转发规则，支持基于域名和 URL 的转发。
- 动态调整转发规则，适配节点变化，自动移除异常节点。
- 保持请求的亲 and 性，支持会话保持或灰度发布。

除了硬件级别的负载均衡，常见的互联网架构主要是使用 4 层和 7 层的负载均衡，常使用的软件负载有 LVS、HAProxy、Nginx 等。LVS 工作在第 4 层，在网络层利用 IP 地址进行转发。Nginx 工作在第 7 层，根据用户的 HTTP 请求（比如根据 URL、消息头）来转发，而 HAProxy 工作在第 4 或 7 层，两者都支持。

数据库及缓存服务

之前最常见数据存储服务的一般指关系型数据库，后来随着互联网业务的发展，逐渐产生了非关系型数据，比如 NoSQL 或 KV 缓存等。关系型数据库提供创建、更新、查询和删除等基本操作，并满足 ACID 的特性，常见的关系型数据库有 MySQL、SQL Server、Oracle 和 PostgreSQL 等数据库，使用最广泛的非 MySQL 莫属。缓存一般是直接将数据放到离计算最近的地方，目前大部分放在内存中，解决 CPU 和 I/O 的速度不匹配的问题，用来加快计算处理速度，通常会对热点数据进行缓存，保证较高的命中率，包括 Memcached 和 Redis，

Redis 在互联网行业中使用最为广泛。数据库及缓存作用如下。

- 提供持久化或非持久化的数据存储的能力。
- 支持操作监控统计，数据的联机备份与恢复，保证高可靠。
- 提供场景化的应用，如缓存提供原生的计数，实现计数服务。

在互联网的架构设计中，数据库及缓存一般相互配合使用来满足不同的场景需求，比如在大流量的请求中会使用缓存来加速，但需要注意缓存预热和缓存穿透等问题。

应用性能管理（APM）

APM 是通过打点或探针技术实现企业的关键应用进行监测，并提供解决的建议和优化措施，甚至可以生成请求拓扑图，能明显提高企业应用的可靠性和质量，提前发现不健康的服务并处理，保证用户得到良好的服务体验，使一个企业的关键业务应用的质量更强。

APM 主要应用在以下方面。

- 服务自运维，监控各种资源指标，比如 CPU、内存、磁盘和网络等。
- 服务框架自省依赖关系统计，前台系统访问路径，慢响应等。
- 根据关键特征进行分布式事务跟踪，实现跨应用的业务追踪。
- 全链路提供从应用性能监测到终端用户体验分析。

小结

云计算给各行各业带来的影响越来越大，从之前的不愿意到现在的主动上“云”，技术人员对云计算的认识也正在发生巨大的改变。无论任何类型的企业，上“云”是必经之路，无非是公有云或私有云区别，因此，企业急切需要根据“云”带来的优势对技术架构进行适当的调整设计来更好地满足业务的发展。在中国，面对非常复杂的技术环境和人口规模的红利，越来越细的业务分工，大规模的稳定系统架构设计和业务需求快速发展的矛盾是技术人员和管理人员需要解决的，接下来的各章节主要围绕这一主题来进行详细的阐述。

第2章 从0到1工程实践

做一个互联网产品，首先就要把项目的基础框架搭好，方便后续的开发。很多产品开始开发的时候，都不太注意规范化，待产品需求越来越复杂，人员越来越多时，整个项目会变得很难维护，甚至会影响产品的持续迭代。没有规范化的构建发布流程，容易导致出错，特别是工程结构越来越复杂时，这个问题会更加明显。所以如果项目一开始，就以工程化的思想去组织代码，以规范化的流程去做构建发布，会给后续的发展打下坚实的基础。

随着项目的发布，至少存在线下、线上两个不同的环境：线下环境用于开发测试，线上环境用于给客户提供服务。两个环境对运行时的依赖（操作系统及其版本、依赖软件及版本等）需求是一致的。在传统发布模式下，需要由类似运维人员这样的角色来维护环境的运行时依赖，各种因素可能会导致虽然需求一致，但实际环境不同，从而为项目的稳定运行带来风险。

本章首先将介绍一些工程化的实践经验，比如如何组织代码、如何对代码进行版本管理及如何规范化地进行构建和发布。然后说明环境不一致会导致的问题，以及如何基于容器技术解决环境不一致问题，并对容器相关的生态系统进行简单介绍。最后结合一个简单的实战示例，给予完整展现。

2.1 工程化

在产品快速迭代、快速发布的情况下，好的工程实践会引导项目向好的方向发展，不至于越来越难维护及扩展。对于产品来说，良好的开发习惯、规范化的测试发布流程可以保证产品的可维护性及质量。

2.1.1 工程模板

要开发一个服务（或者产品），首先就要创建一个工程（或者项目），如何快速生成一

个规范化的项目模板，并且按一定规则组织代码，对于项目后续的维护及演进至关重要。如果一开始不注意规范化，代码组织混乱，就会影响后续的代码维护。

1. 生成代码框架

如果在创建工程的时候，通过工具生成一个简单的代码框架，对开发人员来说就很方便，后续只需要按照框架的规范去填充代码就行。本节以 Java 生态中一个很重要的工具 Maven 来简单说明如何基于 Maven 生成代码框架。

Archetype 简介

Maven 的目标是提供一个简单并且标准化的构建系统，以业界的最佳实践为开发人员的开发构建过程提供参考，并为在不同项目间共享依赖包提供一种标准化的流程和规范。而 Maven Archetype 是 Maven 提供的一个模板工具，通过它可以快速创建一个规范化的项目框架。Archetype 将某一类工程所共有的模式抽象成一个模板，而且会包含实际项目的一些实践经验或者规范，可以帮助开发者快速创建工程框架，并保持良好的风格。

Maven Archetype 提供了很多模板，可以通过 `mvn archetype:generate` 来查看模板列表。基础的模板如表 2-1 所示。

表 2-1 Maven Archetype 基础模板

| Archetype ArtifactIds | 描述 |
|----------------------------|------------------------|
| maven-archetype-quickstart | 用于创建 Maven 示例工程 |
| maven-archetype-webapp | 用于创建一个简单的 Maven Web 工程 |

基于 Archetype 生成框架

我们来看一下如何基于 Archetype 提供的模板生成一个简单的 Maven Web 工程。

```
mvn archetype:generate \  
  -DarchetypeGroupId=org.apache.maven.archetypes \  
  -DarchetypeArtifactId=maven-archetype-webapp \  
  -DgroupId=com.netease.example.shopping \  
  -DartifactId=web \  
  -DinteractiveMode=false
```

基于上述命令，我们就可以生成一个简单的 Web 工程框架，生成的工程目标结构如下。


```
└─ web
    └─ pom.xml
    └─ src
        └─ main
            └─ resources
            └─ webapp
                └─ WEB-INF
                    └─ web.xml
                └─ index.jsp
```

可以看到，工程根目录下有一个 `pom.xml` 文件。POM（Project Object Module，项目对象模型）文件是 Maven 的核心配置文件，它会定义工程的基本属性，如工程名称、版本等，也会定义工程所依赖的组件，构建所需要的一些插件及配置信息等。

`webapp` 目录下会存放一些与 Web 页面相关的内容，如 JSP 文件、静态文件（HTML、JS、CSS 等）、模块文件（FTL 等）等。而 `resources` 目录下会存放一些与配置有关的文件，如数据库配置文件、缓存配置文件、Web 框架配置文件（Spring 的配置文件）等。

2. 代码分层

通过 Archetype 生成一个基础项目框架后，就需要我们去填充代码。在实际的开发过程中，有一些基本的规范去组织代码。

MVC（Model-View-Controller，模型-视图-控制器）是一种设计模式，在开发中，我们也会以此为原则把代码分成 `controller`（控制器）层、`service`（业务逻辑）层、`dao`（数据访问）层和 `meta`（模型）。

接口与实现分离也是日常开发中需要注意的，这个规则背后是面向接口编程的思想。通过接口向外暴露服务，而隐藏实现细节，使用方只通过接口来使用服务，这样既可以保证在某个接口实现变动时，不影响使用方，也可以在需要的时候通过替换接口实现来改变系统行为。比如之前的 Web 工程，按照上述分层原则，添加一些与用户操作相关的代码如下。

```
└─ pom.xml
└─ src
    └─ main
        └─ java
            └─ com
```

```

|       └─ netease
|           └─ example
|               └─ shopping
|                   └─ controller
|                       └─ UserController.java
|                   └─ dao
|                       └─ IUserDao.java
|                           └─ impl
|                               └─ UserDao.java
|                   └─ meta
|                       └─ User.java
|                   └─ service
|                       └─ IUserService.java
|                           └─ impl
|                               └─ UserService.java
└─ resources
    └─ db.properties
└─ webapp
    └─ WEB-INF
        └─ web.xml
    └─ index.jsp

```

我们在其中的 `resources` 目录下放置了数据库访问的配置文件。

3. 单元测试

单元测试是指对软件系统中最小的集合例如方法或函数级别进行测试和验证。单元测试是最低级别的测试活动，这个阶段重点在于验证一个函数的输入输出是否符合设计，而不是验证功能或特性，也不是测试服务提供的接口。单元测试一般都由开发自己完成。

单元测试的目的是在项目早期排查业务代码中比较初级的问题，并且维持系统一个健康的生态圈，在后续不断修改代码的过程中保证最基础的产品质量（例如开发人员不小心修改了前人开发的逻辑导致单元测试失败，就需要排查是真正的设计需要还是不小心引入了问题）。相比有单元测试的项目来说，无单元测试的项目更像是在裸奔，毫无安全感。在单元测试中一般会将测试代码直接集成到业务代码库的方式来维护和运行，使用 Java 语

言并且使用 Maven 工程的业务代码都会有现成的结构以供开发者直接添加，如下所示。

```
├─ pom.xml
├─ src
│   ├── main    ...
│   └─ test
│       ├── java
│       │   └─ com
│       │       └─ netease
│       │           └─ example
│       │               └─ shopping
│       │                   └─ test
│       │                       ├── UserDaoTest.java
│       │                       └─ UserServiceTest.java
│       └─ resources
│           └─ db.properties
```

单元测试一般使用一些用例管理框架来组织自己的测试代码，例如 JUnit、TestNG 等。一般单元测试运行的时候不会真正启动服务，假如业务代码依赖其他模块或者服务，则可以使用 Mock 的方式，尽量做到单元测试执行的时候不依赖其他模块或函数。Mock 的方式有很多，在 Java 中可以用 PowerMock、Mockito 等工具来完成。

2.1.2 模块化

我们了解到如何快速创建一个简单的工程框架来开始简单的产品需求开发。在实际项目中，产品需求一般都比较复杂，功能点也比较多，比如对于一个电商系统，需要有用户管理、商品管理、订单管理和支付等功能，如果没有很好的方式组织这些功能的代码，随着产品需求持续演进，开发人员越来越多，项目的维护成本就会越来越高，甚至影响产品的正常迭代。

1. 什么是模块化

模块化就是将复杂的功能实现拆分到不同的模块中。每个模块功能相对独立，各个模块之间通过定义好的接口沟通。模块拆分遵循高内聚、低耦合的原则，也就是模块内部高内聚，只负责一件事情或者紧密相关的几件事情；模块之间低耦合，相互之间依赖清晰，

只通过接口沟通。

这样做的好处首先是实现了解耦，每个模块只负责一件事情，职责清楚；另一方面，模块化后，每个模块可以由不同的个人或者团队维护，这样更容易维护，也利于各个团队针对模块功能进行持续的演进和优化，而在这个过程中也不会影响其他模块的功能。

模块化可以有不同的层次，比如之前的一个功能对应一个模块，但也可以一个类库对应一个模块，不同的阶段模块化的粒度可能会不同。按功能拆分模块后，可能会有不同的部署方式，可以是代码层拆分，但是部署时还是一个服务，也可以部署时一个功能对应一个服务。本节主要以 Maven 为例介绍代码层面如何拆分及管理多个模块。

2. Maven 多模块管理

Maven 的多模块项目一般都是层级结构，最上层是父模块，父模块管理多个子模块。父模块可以对子模块共享的一些内容进行管理，比如依赖、常量定义、构建的设置等。

在电商网站的示例中，我们可以定义一个 shopping 父模块，然后将用户管理（users）、商品管理（items）、订单管理（orders）、支付（pay）等几个模块定义成子模块，在这个示例中，我们还需要一个向外暴露服务的 web 模块（web）。以此为基础，我们来看一下如何通过 Maven 创建和管理多个模块。

首先创建最上层的 shopping 父模块。

```
mvn archetype:generate \
  -DarchetypeGroupId=org.codehaus.mojo.archetypes \
  -DarchetypeArtifactId=pom-root \
  -DarchetypeVersion=RELEASE \
  -DgroupId=com.netease.example \
  -DartifactId=shopping \
  -DinteractiveMode=false
```

运行完成后，会创建一个 shopping 目录，其中只有一个 POM 文件：

```
.
├── shopping
│   └── pom.xml
```

内容如下。


```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.netease.example.shopping </groupId>
  <artifactId>shopping</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>shopping</name>
</project>
```

然后切换到 `shopping` 目录下 (`cd shopping`)，创建其他几个子模块。

创建 `users` 模块。

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DgroupId=com.netease.example.shopping \
  -DartifactId=users \
  -DinteractiveMode=false
```

创建 `items` 模块。

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DgroupId=com.netease.example.shopping \
  -DartifactId=items \
  -DinteractiveMode=false
```

创建 `orders` 模块。

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DgroupId=com.netease.example.shopping \
  -DartifactId=orders \
  -DinteractiveMode=false
```

创建 pay 模块。

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DgroupId=com.netease.example.shopping \
  -DartifactId=pay \
  -DinteractiveMode=false
```

我们通过 maven-archetype-quickstart 创建成简单的 Maven 工程，而 web 模块则通过 maven-archetype-webapp 创建。

```
mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-webapp \
  -DgroupId=com.netease.example.shopping \
  -DartifactId=web \
  -DinteractiveMode=false
```

最终工程的目标结构如下。

```
.
├── pom.xml
├── items
│   ├── pom.xml
│   └── src
│       ├── main ...
│       └── test ...
├── orders
│   ├── pom.xml
│   └── src
│       ├── main ...
│       └── test ...
├── pay
│   ├── pom.xml
│   └── src
│       └── main ...
```



```

|   └─ test      ...
├─ users
|   └─ pom.xml
|   └─ src
|       └─ main      ...
|       └─ test      ...
└─ web
    └─ pom.xml
    └─ src
        └─ main
            └─ resources
            └─ webapp
                └─ WEB-INF
                    └─ web.xml
                    └─ index.jsp

```

我们再来看看之前生成在 shopping 目录下的 POM 文件。

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/
POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.netease.example.shopping</groupId>
  <artifactId>shopping</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>shopping</name>
  <modules>
    <module>users</module>
    <module>items</module>
    <module>orders</module>
    <module>pay</module>
    <module>web</module>
  </modules>
</project>

```

可以看到其中已经添加了 `users`、`items`、`orders`、`pay` 和 `web` 几个模块。模块之间也会存在依赖，比如这里的 `web` 模块需要依赖其他几个模块提供的接口，就需要在 `web` 模块的 POM 文件中声明对其他几个模块的依赖。

2.1.3 工程化构建

有了基础的代码框架后，开发人员要做的就是根据产品需求来填充代码。功能开发完毕后，最重要的就是构建、打包和部署。规范化，甚至自动化这一流程对于工程的整体开发测试流程来说都是很有必要的，它可以减少错误或者不必要的麻烦。并且随着产品的发展，快速迭代、快速发布会成为产品研发的日常需要，如果没有一个规范或者自动化的流程，那么对于运维人员来说，产品发布就会成为梦魇。本节以 `Maven` 为例，对工程的依赖管理及开发完成后的构建部署流程进行简单的介绍。

1. 基于 Maven 构建

依赖管理 (Dependencies)

一般的工程都需要依赖很多组件去完成功能，比如说依赖 `Web` 开发框架，依赖数据访问框架、日志框架等。对一个简单工程来说，依赖管理可以很简单，但对一个多人维护的大型项目来说，依赖管理就会很复杂。之前提到 `Maven` 时说过，`Maven` 会提供一种标准化的依赖管理机制，让大型项目的依赖管理很简单。`Maven` 的依赖管理既可以管理项目对于外部框架的依赖，也可以管理工程本身内部各模块之间的依赖。以下是一个简单的示例。

```
<project> ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
```

```
<scope>provided</scope>
</dependency>
</dependencies> ...
</project>
```

其中 `dependency` 元素有一个 `scope` 的配置, `scope` 控制哪些依赖在什么环境下可用或者哪些依赖在打包时需要 (或者不需要)。下面介绍一下 Maven 里常用的几种 `scope`。

- **compile**: 默认的 `scope`, 这种 `scope` 的依赖在所有场景下都可以, 打包时也会打包进应用。
- **provided**: 预期由 JDK 或者运行时的容器来提供的依赖, 可以使用此 `scope`, 比如上面示例中的 `servlet-api` 就由运行的 `Servlet` 容器提供, 打包时不需要提供这个依赖 (但是编译时需要)。
- **runtime**: 在测试或者运行的时候需要依赖, 但在编译的时候不需要。
- **test**: 只在编译或者运行与测试相关的代码时需要。

Maven 会在构建的时候从远程 Maven 仓库下载依赖到本地 (如果已经下载到本地, 则直接从本地加载依赖), 并完成构建。从网络上自动下载依赖革命性地改变了开发软件的方式。

构建生命周期 (Lifecycle)

Maven 的构建动作是由构建目标 (goal) 与构建目标的不同阶段 (phase) 组成的。构建生命周期就是指构建过程中的不同阶段。Maven 中的构建工程都需要遵循生命周期, 它的默认生命周期包含常用的工程构建阶段, 如编译、测试、打包等。以下是 Maven 默认生命周期常用各个阶段的简要说明。

- **compile**: 对工程的源代码进行编译。
- **test**: 基于单元测试框架运行所有的单元测试。
- **package**: 对编译后的代码进行打包, 比如 JAR、WAR 等。
- **install**: 将打包好的文件安装到本地 Maven 仓库。
- **deploy**: 将打包好的文件安装到远程 Maven 仓库。

构建 Profile

为了解决程序可移植性的问题，Maven 2.0 引入了 Profile 的概念。基于 Profile，Maven 才有了可以针对不同环境进行不同构建的能力。比如对于线上线下环境，一般都会访问不同的数据库，通过不同的 Profile 来完成：针对线上环境配置线上数据库地址，针对线下环境配置线下数据库地址，在构建时指定是针对线上环境构建还是线下环境构建。Profile 也可以配置成基于操作系统或者安装的 JDK 版本触发不同的构建 Profile，这样在不同环境下构建出来的应用就不会存在兼容性问题。

2. 构建示例

有了上面对 Maven 构建相关内容的了解，我们就可以基于 Maven 进行构建，例如针对下面的配置。

```
<project> ...
  <profiles>
    <profile>
      <id>online</id>
      <properties>
        <db.url>ip1:port</db.url>
      </properties>
    </profile>
    <profile>
      <id>test</id>
      <properties>
        <db.url>ip2:port</db.url>
      </properties>
    </profile>
  </profiles>
</project>
```

online 和 test 两个 Profile 访问的数据库地址不同，我们可以通过如下的命令针对线上环境打包应用。

```
mvn package -Ponline
```

如果开发一个框架性的项目，就可以通过 `mvn install` 将生成的依赖安装到本地 Maven

仓库,如果有其他团队依赖这个框架,就可以通过 `mvn deploy` 将依赖包部署到远程 Maven 仓库。

多模块 Maven 工程的构建与单模块的差不多,只不过多模块工程在构建时,需要分析各个模块之间的依赖,比如电商网站示例里, `web` 依赖其他几个模块提供接口,构建时就先构建其他几个模块,然后再构建 `web` 模块。

2.1.4 代码规范及检查

1. 代码规范

代码规范有两方面的内容,一方面是代码风格,如使用 Tab 还是空格,缩进 2 格还是 4 格,对变量或者函数命名时,是使用驼峰法还是使用下划线命名法;另一方面是代码实现层面的一些基础规范,比如异常怎么处理、日志怎么处理。

有些读者会纳闷,只要我完成功能就好了,为什么要花时间在这些事情上?难道差 2 个空格差别就这么大?或者按自己的想法处理一下异常就好了,又会有什么影响?

对于一个产品来说,一般都由一个团队来完成,如果没有统一的代码规范,那么每个人的代码必定风格迥异。如果多个人同时开发同一模块,或者即使是模块分工明确,合并代码的时候也会碰到问题。即使你没改任何代码,但是格式不一致(或者开发工具做了格式化),合并后还是会有代码更新,而实际可能只是因为 Tab 变成了空格。一般情况下,理解代码困难并非是因为代码中有复杂的算法或者逻辑,而是不习惯阅读风格不一致的代码,统一的风格使得代码可读性提高。而没有规范的异常处理,没有规范的日志处理等,就可能在排查问题时很难找到引起问题的原因或者效率低下。基于经过实践验证的代码规范进行开发,不但可以有效减少问题,查找问题也会更有效率。

随着项目的推进,维护成本成为项目的主要问题。而开发过程中的代码质量直接影响着维护的成本。上面提到,规范的代码大大提高了程序的可读性,而可读性高的代码维护成本相对更低。维护工作不仅要读懂原有代码,还需要在原有代码基础上做修改,统一的风格有利于长期的维护。

所以,规范的代码在团队的合作开发中是非常有益而且必要的,它的主要目的是代码可读性和可维护性。每个团队都应该有自己的代码规范,参考一些大公司的代码规范,在其基础上整合团队的特殊性,并持续改进,以提高团队整体的代码质量。

2. 代码检查

如果每个团队都要求你在写出一行代码的时候注意风格，那就得不偿失。基本上主流语言的开发工具都提供了代码格式化的工具，通过定义规则，开发工具可以在保存时自动（或者手动）对代码进行格式化。

当然，除了代码格式化工具，也可以在代码提交时（或者合并前），通过一些工具（如 Java 中的 Checkstyle）对代码进行强制代码风格检查，即使就算人为原因没做格式化，代码也会因为检查出问题而无法提交或合并。

代码风格检查只保证了代码风格的一致，但实现是否合理，是不是有哪些隐藏的问题没有发现，我们往往需要一些工具来进行更严格的检查。这就是静态代码检查工具想达到的目标。静态代码检查会分析所有代码，来发现所有可能的问题，比如潜在的空指针，资源没有关闭或者回收，可以优化的代码等。Java 可以通过 Findbugs 或者 PMD 来做静态代码检查。

2.1.5 代码版本管理

设想这样的场景，你对代码做了一些变更，上线后发现代码有问题无法正常运行怎么办？你肯定会想赶紧回滚到原来的代码，如果没有代码版本控制的工具，就很麻烦。

代码版本管理系统是一种可以帮助开发团队管理代码变更的工具，它会跟踪团队成员对代码的每一次修改。如果发现变更的代码有问题，开发者可以回滚代码，并且可以与之前的代码对比，找出问题所在，尽量减小对整个团队成员的影响。对于开发团队来说，代码是最重要的资产，代码版本管理工具可以避免人为或其他原因导致的代码丢失或文件损坏。

1. 代码管理工具

代码版本管理工具经过这么多年的发展，目前主要有两种不同的模式：集中式版本控制（Centralized Version Control Systems，简称 CVCS）及分布式版本控制（Distributed Version Control System，简称 DVCS）。

集中式版本控制

Subversion（简称 SVN）是集中式版本控制工具的典型代表。集中式版本控制工具有

一个单一的集中管理服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。这种做法带来了许多好处，每个人都可以在一定程度上看到项目中的其他人正在做些什么，管理员也可以轻松掌控每个开发者的权限。

事分两面，有好有坏。这么做最显而易见的缺点是中央服务器的单点故障。如果宕机一小时，那么在这一小时内，谁都无法提交更新，也就无法协同工作。如果中央服务器的磁盘发生故障，碰巧没做备份，或者备份不够及时，就会有丢失数据的风险。最坏的情况是彻底丢失整个项目的全部历史更改记录，而被客户端偶然提取出来保存在本地的某些快照数据就成了恢复数据的希望。但这样依然是个问题，你不能保证所有的数据都事先完整提取出来。

分布式版本控制

因为集中式版本控制的单点问题，分布式版本控制系统面世了，Git 是其代表。在分布式版本控制中，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。因为每一次的提取操作，实际上都是对代码仓库的完整备份。

许多这类系统都可以指定和若干不同的远端代码仓库进行交互，你可以在同一个项目中，分别和不同工作小组的人相互协作。

2. Git VS. SVN

这两种代码管理工具的基本使用方式在此不做详细介绍，读者可以分别参考 *Pro Git Book* 及 *SVN 教程*。

不管是 Git 还是 SVN，一般的使用流程都是检出 (checkout)、添加或者更新文件 (add)、提交变更 (commit) 的过程，只是内在的一些概念有一些区别。

在 SVN 中，只有一个远程中央仓库，所有的操作都针对这个远程仓库。而在 Git 中，既有远程仓库，又有本地仓库的概念，本地仓库可以同步远程仓库的更新，但本地仓库独立于开发者，每个开发者都可以独立操作自己的本地仓库而不影响其他开发者。Git 中有很多操作都是针对本地仓库，确定没问题时再同步到远程仓库。

SVN 的 commit 直接提交代码到远程仓库，如果发现提交的代码有问题，就再提交一

次；而 Git 的 commit 只提交到本地仓库，需要通过 push 才能将提交推送到远程仓库，如果在 push 前发现本地提交有问题，还可以通过 revert、rebase 等操作去变更提交记录。

在 SVN 中，分支（branch）是一个完整的目录，包含完整的实际文件，新建一个分支就等于把所有文件复制一份，所以在 SVN 中，创建分支是一件相对来说有代价的事（耗时复制，新的复制占用空间）。而且这个分支是大家共享的，你新开一个分支，其他成员就可以看到（所以 SVN 有目录级权限控制，以控制开发人员对不同目录的访问权限）。

在 Git 中，创建分支不需要复制文件，它基于提交创建分支，所以 Git 创建分支非常快。Git 首先是在本地仓库中创建分支的，只有自己能够看到，只有在确定这个分支需要共享给其他人时，才会把分支推送到远程仓库。这些差异，也是 Git 分支的优势，Git 鼓励多分支，通过分支来组织不同的功能开发。

3. 分支模型

当我们多人协作开发的时候，往往会产生多个分支，因此一个好的分支管理策略就显得比较重要，而一个混乱的分支管理，可能会导致提交历史看上去一团糟。下面以 Git 为例，介绍两种分支模型，并结合笔者的实际开发体验说明差异。

master/develop 多分支模型

如果读者在网上搜索 Git 的分支模型，会看到来自如下地址的一篇文章 <http://nvie.com/posts/a-successful-git-branching-model/>，在这篇文章中，作者 Vincent Driessen 提出了自己使用的一个 Git 分支模型，并且非常成功。

如图 2-1 所示，这个分支模型中使用两个长期分支 master 和 develop，然后使用多个辅助分支。当进行新功能开发的时候，需要首先从 develop 分支中开新的特性分支进行开发，当一个新的特性开发完成后，然后合并回 develop 分支；当开发稳定后，可以发布一个新版本的时候，从 develop 分支开一个 release 分支，再在 release 分支上面进行一些提交，待稳定后，合并到 master 分支发布，同时相关的 release 分支上的修改还要合并回 develop 分支；当线上出现问题后，直接从 master 上面开 hotfix 分支，待完成修复后，再合并到 master 分支，最后相关的修复还要合并到 develop 分支。该分支模型的最大特点就是使用显式的合并来跟踪新特性的合入、新版本的发布，以及线上的 hotfix。当发现某个新特性的合入导致问题时，可以仅仅恢复那个合并的提交，即可将所有与该特性相关的提交进行恢复。

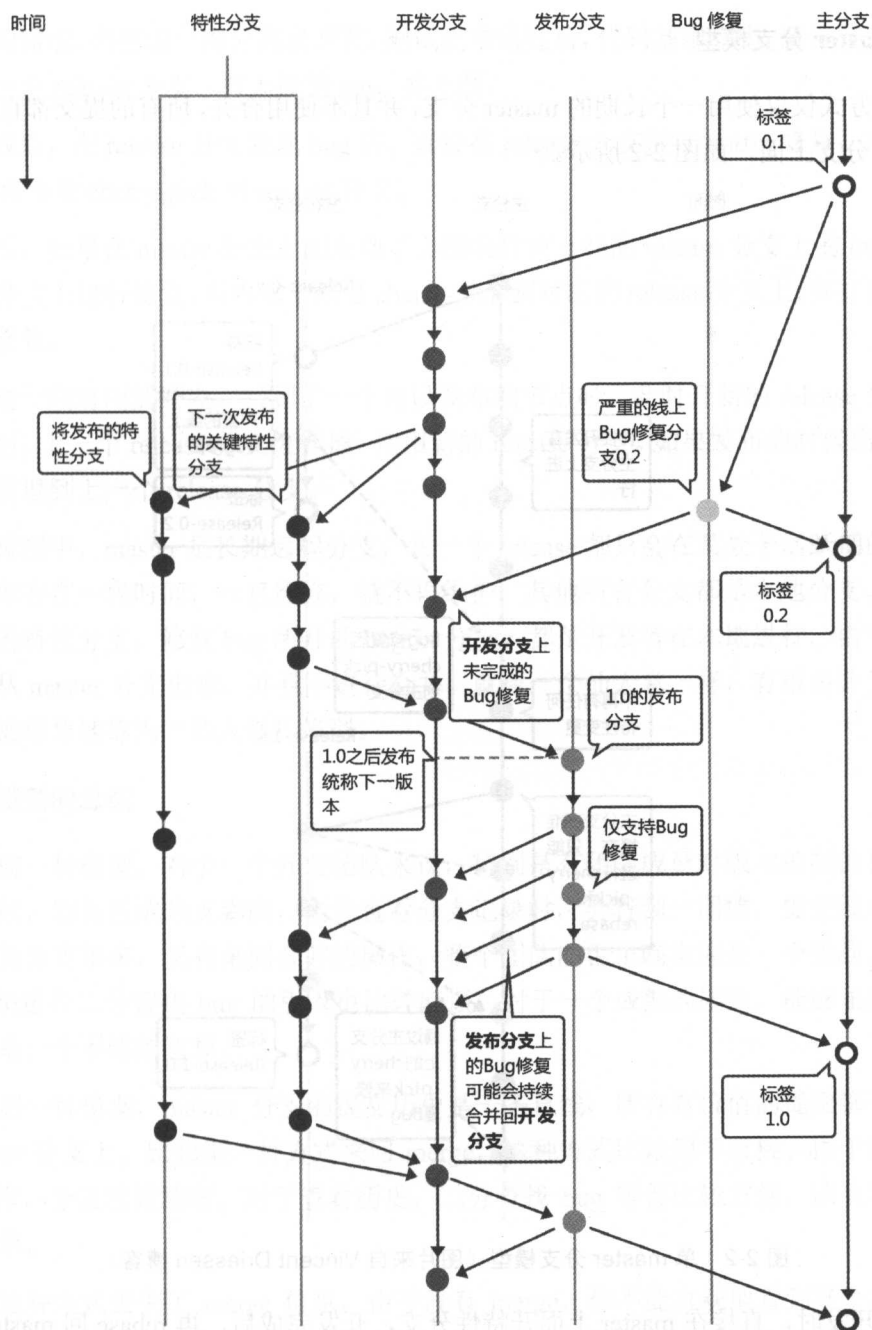


图 2-1 多分支模型 (图片来自 Vincent Driessen 博客)

单 master 分支模型

这种方式仅仅使用一个长期的 master 分支，并且不使用合并，所有的提交都直接 rebase 到 master 分支上面，如图 2-2 所示。

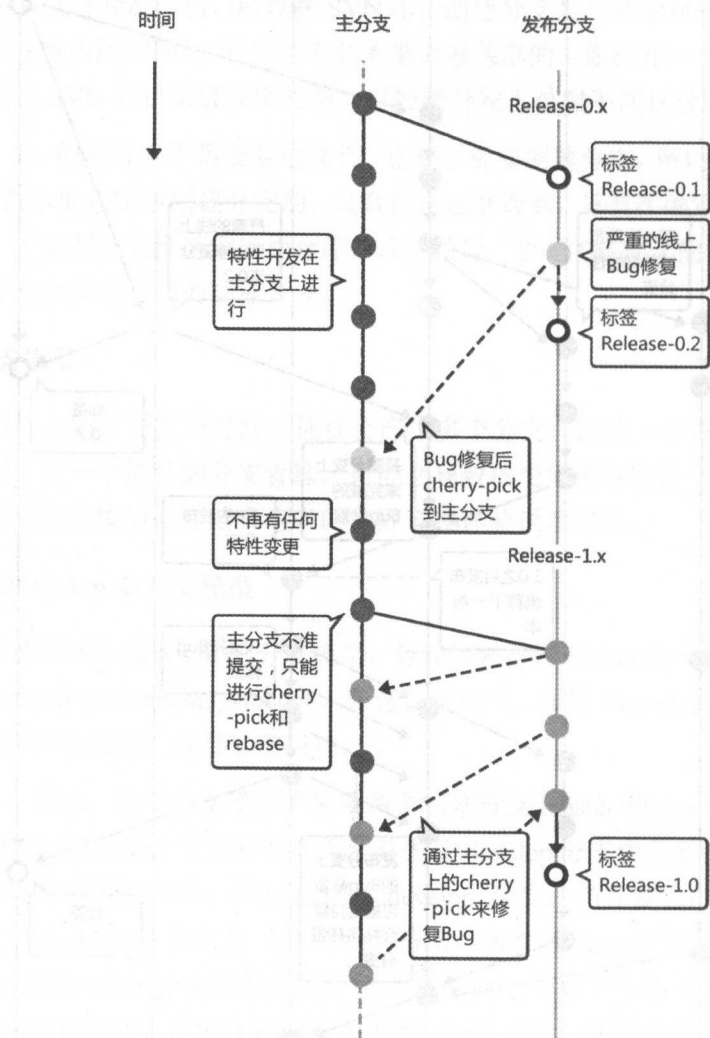


图 2-2 单 master 分支模型（图片来自 Vincent Driessen 博客）

功能开发时，直接在 master 上面开特性分支，开发完成后，再 rebase 回 master 分支。如果需要 bugfix，也同样直接在 master 上进行修复，完成后再直接 rebase 到 master 分支。

根据具体的需求,当完成一部分需求开发,测试正常通过后,代码基本稳定,就直接从 `master` 分支开出一个 `release` 分支,打上线的 `tag`,并上线。

当上线后,在 `release` 分支发现 `bug` 后,直接在 `release` 分支进行 `hotfix` 修复,并打新的 `tag`,然后将修复 `cherry-pick` 回 `master` 分支。

上线后,如果在 `master` 分支上面发现了会影响目前上线的 `release` 分支上的 `bug`,直接在 `master` 分支上进行修复,再将这个修复 `cherry-pick` 到对应的 `release` 分支上,并打新的 `tag`,然后上线修复。

再经过一段时间的开发,又到了可以发布的节点时,此时开新的 `release` 分支。当发布完成后,上一个 `release` 分支废弃掉,使用新的 `release` 分支。如果发布的时候出现问题,可以直接回退到上一个 `release` 分支。

这种模型中, `master` 是长期远程分支,每一个 `release` 都只会在其处于活跃期的时候,在远程仓库存在一段时间,一旦废弃,就不再维护。其他所有分支都是本地分支,包括开发新功能的特性分支,修复 `bug` 所开的 `bugfix` 分支,都是开发者在本地进行。由于其所有分支都是从 `master` 分支出来,并且不进行合并,就像一个仙人掌一样,有很多分叉,因此该模型又被形象地称为“仙人掌”模型。

两种模型的总结

使用前一种模型,对于一个开发团队来说,特别是有团队成员对版本控制的使用不太熟练的时候,容易造成分支膨胀,当你查看分支记录时,会看到一团糟,提交线混乱。而且由于涉及分支很多,又有来回合并的操作,整个团队能否正确实施是一个挑战。一旦出现 `bug`,想进行二分查找 `bug` 的引入也比较麻烦。对于一个成熟的团队,能够正确实施该模型,也是一个不错的选择。

使用后一种模型, `master` 分支的提交历史是一条直线,所有有价值的提交都一定要应用到 `master` 分支上。比起第一种方式来回 `merge`,这种方式比较简单直接。由于限制使用 `merge` 操作,分支比较清晰。对于查看历史,二分查找 `bug` 等都比较方便,团队实施起来也比较简单。

不过这种方式丢失了 `merge` 信息,由于没有 `merge`,你不能直观地看到某个功能在哪里进入到 `master` 中,不过可以通过强制在提交日志中说明其实现的功能来弥补。

4. 第三方托管工具

如果你不想自己来维护一套代码管理或者镜像管理的系统，可以使用第三方托管平台。一般只需要注册一个帐号，就可以创建私有或者公开的仓库，方便快速地进行代码或者镜像管理。

GitHub (<https://github.com/>)

GitHub 是一个面向开源及私有软件项目的托管平台，因为只支持 Git 作为唯一的版本库格式进行托管，故名 GitHub。

GitHub 于 2008 年 4 月 10 日正式上线，除了 Git 代码仓库托管及基本的 Web 管理界面以外，还提供了订阅、讨论组、文本渲染、在线文件编辑器、协作图谱（报表）、代码片段分享（Gist）等功能。目前，其注册用户已经超过百万，托管版本数量也非常多。

GitHub 同时提供付费账户和免费账户。这两种账户都可以创建公开的代码仓库，但是付费账户也可以创建私有的代码仓库。除了允许个人、组织创建和访问代码库以外，它也提供了一些方便社会化软件开发的功能，包括允许用户追踪其他用户、组织、软件库的动态，对软件代码的改动和 bug 提出评论等。GitHub 也提供了图表功能，用于显示开发者们怎样在代码库上工作及软件的开发活跃程度。

GitLab (<https://gitlab.com/>)

GitLab 是一个利用 Ruby on Rails 开发的开源应用程序，实现一个自托管的 Git 项目仓库，可通过 Web 界面进行访问的公开或者私人项目。

它拥有与 GitHub 类似的功能，能够浏览源代码、管理缺陷和注释。可以管理团队对仓库的访问，易于浏览已提交的版本并提供一个文件历史库。它还提供一个代码片段收集功能，轻松实现代码复用，便于日后有需要的时候进行查找。

相比 GitHub，GitLab 上可以免费创建自己的私有项目。

码云 (<http://git.oschina.net/>)

码云是开源中国社区于 2013 年推出的基于 Git 的完全免费的代码托管服务，这个服务是基于 GitLab 开源软件所开发的，在 GitLab 的基础上做了大量的改进和定制开发，目前已经成为国内最大的代码托管系统，致力于为国内开发者提供优质稳定的托管服务。

码云除了提供最基础的 Git 代码托管之外,还提供代码在线查看、历史版本查看、Fork、Pull Request、打包下载任意版本、Issue、Wiki、保护分支、代码质量检测和 PaaS 项目演示等方便管理、开发、协作、共享的功能。

2.1.6 环境划分

在产品开发过程中,为了快速开发,根据不同目的,通常我们都会部署多个环境以完成不同目的。下面介绍一种常见的环境划分方式。

开发环境

开发环境是用来给开发者开发时调试及测试使用的,一般不对性能容量之类的做要求。开发人员开发完成后,需要根据测试人员给出测试用例进行基本的冒烟测试,这个冒烟测试一般在开发环境进行,这个环境由开发人员管理,代码一有更新,就可以随时更新这些环境。注意,这里的开发环境不是指开发人员的本地开发环境。

测试环境

测试环境主要是给测试人员使用的。它的主要目的是给测试人员一个相对可控的环境,不会因为代码更新等原因就更新环境,从而不会导致测试人员需要不断回归已经测试的用例是否会因为新进的代码而产生问题。测试环境由测试人员管理,一般都是一轮测试完成后,开发人员对相关问题进行修复,然后由测试人员进行更新。

集成测试环境

一般产品到一定阶段后,都会由多个模块组成,模块之间相互依赖。开发者在开发环境或者测试人员在测试环境中,更多的是关注模块内部的功能点,甚至在需要的时候,会去 Mock 所依赖的其他模块提供的服务,从而不至于因为其他模块的开发进度影响到本模块的开发测试进度。而集成测试环境,主要目的是确保各个模块集成起来后的功能是否符合整体产品的预期,模块之间的沟通是否符合之前定义好的接口规范。

预发布环境

当所有功能通过测试人员的测试后,可以在上线到生产环境前,发布到预发布环境。预发布环境一般与生产环境一致,主要目的是在正式上线前对要发布的版本做一次上线演

练，减少正式上线的风险。上线到预发布环境后，除了开发测试人员，其他非技术人员，如产品、交互、视觉等就可以在这些环境中对要上线的版本进行体验。

生产（线上）环境

生产环境是线上正式运行的环境，要求所有功能都通过测试并保持稳定。

环境一致性问题

应用对不同运行环境（比如需要哪些软件等）的需求一般需要通过系统管理员之类的角色来做变更，不同环境（开发、测试、生产）的变更都是独立进行的。一个典型的开发过程，首先是开发者根据开发需求，在开发环境配置各种环境依赖；开发完成后，然后根据我们的环境需求配置测试环境；最后测试联调完成后，配置线上生产环境，完成上线过程。在这个过程中，各个环境的配置是相对独立的，虽然环境需求一致，但在切换到一个全新的环境时，按照依赖需要安排各种依赖，在这个过程中，软硬件架构等原因导致虽然依赖的软件一致，但实际的实现或者某些细节不一致，就会对应用的稳定运行带来风险。

容器世界中的 Docker 镜像就可以解决此类问题，它可以像代码管理工具一样，让你管理代码的运行环境。通过把运行环境镜像化，就可以在开发测试及上线过程中都使用同一套镜像，并且这个镜像可以由开发人员完全把控，开发人员只要在开发阶段确定了开发所需要的运行镜像，后续的流程都基于此镜像进行，保证运行环境的完全一致性。同时，镜像也支持版本化，如果你的镜像因为环境变更而影响了程序运行，可以轻松回滚到这个镜像的前一个版本。

2.2 基于容器工程化

Docker 容器可以在不同的开发与产品发布生命周期中保证一致性，进而标准化你的环境。除此之外，Docker 镜像还可以像 Git 仓库一样，让你提交变更到镜像中并通过不同的版本来管理它们。设想如果你因为完成了一个组件的升级而导致整个环境都损坏了，可以轻松回滚到这个镜像的前一个版本。整个过程只要在几秒内完成，就能满足产品对于快速迭代及快速发布回滚的需求。

2.2.1 Docker 及作用

容器不是什么新技术，只是隔离及封装操作系统资源的一种方式。技术潮起潮落，往往革命性的技术都是被人们关注已久的旧技术，所谓“新瓶装旧酒”，风味尤佳！容器技术由来已久，历史上的容器技术如表 2-2 所示。

表 2-2 历史上的容器技术

| 容器技术名称 | 操作系统 | 遵循协议 | 发行时间（年） |
|--------------------|---------|-------------|---------|
| chroot | *NIX | Varies | 1982 |
| FreeBSD Jail | FreeBSD | BSD License | 1998 |
| Linux-VServer | Linux | GNU GPLv2 | 2001 |
| Solaris Containers | Solaris | CDDL2004 | 2004 |
| OpenVZ | Linux | GNU GPL v.2 | 2005 |
| Virtuozzo | Windows | Proprietary | 2007 |
| LXC | Linux | GNU GPLv2 | 2008 |
| Warden | Linux | Apache 2.0 | 2012 |
| Docker | Linux | Apache 2.0 | 2013 |

从表 2-2 可以看出，Docker 并不是最新的容器技术，但从一定意义上来说，Docker 开创了 Linux 容器界的文艺复兴时代，人们对 Docker 的关注持续增长，不断发掘适用场景，使得容器技术得到广泛的认知和应用。

创新并不意味着要创造出全新技术，借助 Docker 的势头，容器技术一跃成为一种颠覆性的技术，它能改变 IT 行业的格局、相关的生态系统和云计算市场。

1. Docker 简介

Docker 的诞生

2013 年 3 月 15 日，在加利福尼亚州圣克拉拉市举办的 Python 开发者大会上，dotCloud 的创始人兼 CEO Solomon Hykes 在一个短短 5 分钟的微型演讲“The future of Linux Containers”中向世人宣布了 Docker，并当场向观众演示了 Docker 的 Hello World 示例（参见安装 Docker 环境）。宣布 Docker 时，在 dotCloud 公司之外只有 40 人有机会使用 Docker。

但就在宣布之后的几周之内，Docker 得到了广泛的报道，很快 Docker 开源了，源码托

管在 GitHub 中，任何人都可以下载或者为这个项目做贡献。在接下来的几个月里，业界听说 Docker 的人越来越多。从此，Docker 带动了一股容器浪潮，引爆了容器市场。直至今今天，业界几乎没有人不知道 Docker。但还是有一些人不确定 Docker 是什么，它能做什么？下文就为读者“拨开云雾见天日”。

一句话解释 Docker

Docker 是一个开源的容器引擎，提供了一套完整的容器解决方案。Docker 经历了积淀、变革及进化 3 个过程。容器是一种历史悠久的虚拟化技术，一个容器实质上就是运行在宿主机上的一个进程。容器的最小组成公式如下。

容器 = cgroups + namespaces + rootfs + engine

其中，

- cgroups 是资源控制。
- namespaces 是访问隔离。
- rootfs 是文件系统。
- engine 是容器引擎，控制容器生命周期。

Docker 一直将自己比作集装箱，表 2-3 通过和集装箱对比，解析 Docker 的特性。

表 2-3 Docker 与物理集装箱的相似性

| / | 物理集装箱 | Docker |
|-----------|--|---|
| 无关内容 | 集装箱几乎支持所有类型的货物 | 能压缩任何负载和依赖 |
| 无关硬件 | 标准化的尺寸和交接使得集装箱可以通过货船火车货车运输，用起重机交接不需要换一个容器或者打开集装箱 | 使用操作系统的基本体（像 LXC）能一致运行在几乎任何硬件下，不需要对硬件做额外的修改 |
| 内容隔离和相互作用 | 不需要关心铁锹会砸坏香蕉。集装箱可以一起堆放一起运输 | 资源、网络和内容的独立避免了依赖问题 |
| 自动化 | 标准化使得自动化装载卸货和移动变得方便 | 使用标准的操作指令去运行 start/stop/commit/search 等 |
| 高效 | 不需要打开，不需要改动什么，高效的点对点方式 | 轻量级，几乎没有启动的消耗，高效的移植性和操控性 |
| 职责分离 | 发货人只需要关心箱子内部的事情，托运人只需要关心箱子外部的的事情 | 开发者只需要关心代码层面，运营人员只需要关心服务器的基础环境 |

Docker 容器的优势

Docker 容器的优势，和虚拟机（VM）相比较尤为显著，如表 2-4 所示。

表 2-4 Docker 容器与虚拟机的比较

| / | Docker 容器 | 虚拟机（VM） |
|-------|-----------------|------------------|
| 操作系统 | 与宿主机共享 OS | 宿主机 OS 上运行虚拟机 OS |
| 存储大小 | 镜像小，便于存储与传输 | 镜像庞大（vmdk、vdi 等） |
| 运行性能 | 几乎无额外性能损失 | 操作系统额外的 CPU、内存消耗 |
| 移植性 | 轻便、灵活，适应于 Linux | 笨重，与虚拟化技术耦合度高 |
| 硬件亲和性 | 面向软件开发者 | 面向硬件运维人员 |
| 部署速度 | 快速，秒级 | 较慢，10s 以上 |

总的来说，Docker 更轻量、更快速、更便捷，它是一套以容器技术为核心的，用于应用的构建、发布和执行的体系和生态。

2. 安装 Docker 环境

随着 Docker 工具链的完善，在各个平台上安装 Docker 非常容易。

Linux

Docker 原生支持大多数 64 位 Linux，大多数流行的 Linux 发行版都源自 Debian 或 Red Hat，安装起来大同小异，本节会以常用的 Debian 和 CentOS 为例演示安装，如果你有在其他 Linux 下安装 Docker 的需求，可以访问 Linux 安装的官方网址：Install Docker on Linux distributions。

另外，在 Linux 下也可以使用官方提供的脚本来安装 Docker，一键安装，简单快捷！

```
$ sudo curl -sSL https://get.docker.com/ | sh
```

Mac 和 Windows

对 Mac OS X 和 Microsoft Windows 而言，最推荐的安装方式莫过于 Docker 官方提供的 Native 应用安装包了。随着 Docker 的发展，安装过程也越来越方便，官网文档链接的讲解已经非常详细，这里就不赘述。

Mac 的 Docker 安装: <https://docs.docker.com/docker-for-mac/install/>

Windows 的 Docker 安装: <https://docs.docker.com/docker-for-windows/install/>

2.2.2 Docker 镜像及操作

Docker 包含 3 个基本概念, 分别是镜像(Image)、容器(Container)和仓库(Repository)。镜像 Docker 运行容器的前提, 仓库是存放镜像的场所, 可见镜像更是 Docker 的核心。

镜像作为 Docker 最突出的创新之一, 它变革了软件交付标准。理解镜像, 对理解整个 Docker 的生命周期非常重要。本节将围绕镜像这一核心概念, 简明扼要地介绍镜像的各种操作, 带你玩转 Docker 镜像。

注: 对镜像应用感兴趣的读者, 还可以查看网易云推出的“玩转 Docker 镜像”系列图文和视频。

1. 镜像层 (Layers)

在镜像操作之前, 首先了解下镜像的原理, 其中最重要的概念就是镜像层 (Layers)。镜像层依赖于一系列的底层技术, 比如文件系统 (filesystems)、写时复制 (copy-on-write)、联合挂载 (union mounts) 等, 如图 2-3 所示。读者可以在很多地方学习到这些技术, 见 <https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/>, 这里就不再赘述技术细节。

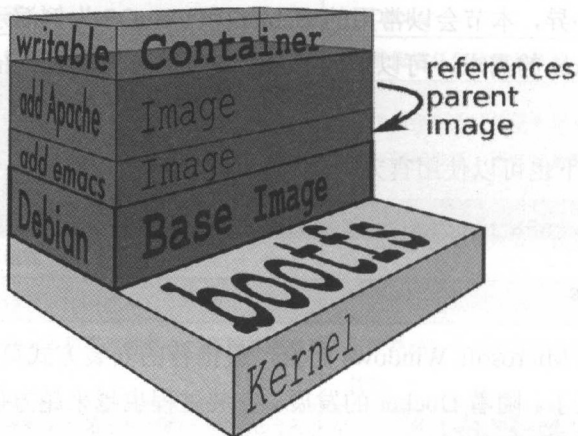


图 2-3 Docker 镜像的典型结构 (图片来自 Docker 社区)

总的来说，最需要记住的一点是：在创建镜像时，每一条指令都会创建一个镜像层，继而增加整体镜像的大小。

2. 获取镜像

可以使用 `docker pull` 从仓库注册中心获取所需要的镜像。

用法如下。

| Usage: docker pull | | | [OPTIONS] | NAME[:TAG@DIGEST] |
|--------------------|------|---------|-----------|-------------------|
| 举例来说 | | | | |
| docker | pull | busybox | | |

以上是从 Docker Hub（Docker 官方镜像仓库）中下载 tag 为 latest（默认）的 BusyBox（一个精简的 Unix 工具集）镜像，同样你也可以从网易云的镜像中心中下载 tag 为 1.25 的镜像。

```
docker pull hub.c.163.com/library/busybox:1.25
```

注：想要终止 `docker pull`，按下 `CTRL+C` 组合键即可，更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/pull/>。

3. 查看镜像

使用 `docker images` 可以列出主机上已有的镜像。

用法如下。

```
Usage: docker images [OPTIONS] [REPOSITORY[:TAG]]
```

举例如下。

```
$ docker images
REPOSITORY      TAG              IMAGE ID         CREATED          SIZE
<none>          <none>          77af4d6b9913    19 hours ago    1.089 GB
committ         latest          b6fa739cedf5    19 hours ago    1.089 GB
```

指定仓库名和镜像。

```
$ docker images java:8
REPOSITORY      TAG              IMAGE ID         CREATED          SIZE
java            8               308e519aac60    6 days ago      824.5 MB
```

更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/images/>。

4. 创建镜像

有两种常用的方法来创建镜像。

- 基于已有镜像的容器创建，使用 `docker commit` 命令。
- 基于 `Dockerfile` 创建，使用 `docker build` 命令。

我们推荐第二种方式，因为 `Dockerfile` 能记录整个镜像的创建过程，后续的章节中将重点介绍。

`Dockerfile` 使用细节请参考：<https://docs.docker.com/engine/reference/builder/>。

基于已有镜像的容器创建用法如下。

```
Usage: docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

举例如下。

```
$ docker ps
ID          IMAGE          COMMAND        CREATED        STATUS        PORTS
c3f279d17e0a  ubuntu:12.04   /bin/bash      7 days ago     Up 25 hours
197387f1b436  ubuntu:12.04   /bin/bash      7 days ago     Up 25 hours
$ docker commit c3f279d17e0a svendowideit/testimage:version3 f5283438590d
$ docker images
REPOSITORY          TAG          ID          CREATED        SIZE
svendowideit/testimage  version3    f5283438590d  16 seconds ago  335.7 MB
```

更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/commit/>。

5. 删除镜像

使用 `docker rmi` 来删除一个或多个镜像，可以通过“镜像名：标签名”来删除，也可以通过“镜像 ID”来删除。

用法如下。

```
Usage: docker rmi [OPTIONS] IMAGE [IMAGE...]
```

举例如下。

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test1               latest             fd484f19954f       23 seconds ago     7 B (virtual 4.964 MB)
test                latest             fd484f19954f       23 seconds ago     7 B (virtual 4.964 MB)
test2               1.0                fd484f19954f       23 seconds ago     7 B (virtual 4.964 MB)
$ docker rmi fd484f19954f
Error: Conflict, cannot delete image fd484f19954f because it is tagged in
multiple repositories, use -f to force
2013/12/11 05:47:16 Error: failed to remove one or more images
$ docker rmi test1
Untagged: test1:latest
$ docker rmi test2:1.0
Untagged: test2:1.0
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test1               latest             fd484f19954f       23 seconds ago     7 B (virtual 4.964 MB)
$ docker rmi test
Untagged: test:latest
Deleted: fd484f19954f4920da7ff372b5067f5b7ddb2fd3830cecd17b96ea9e286ba5b8
```

注：（1）可以通过 `-f` 选项来强行删除镜像，但是不推荐，因为这样往往会造成一些残留问题，比如增加临时镜像。

（2）正确的做法是，先删除依赖该镜像的所有容器，再来删除镜像。

更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/images/>。

6. 搜索镜像

使用 `docker search` 可以搜索远端仓库中共享的镜像。

用法如下。

```
Usage: docker search [OPTIONS] TERM
```

举例如下。

```
$ docker search busybox
```

| NAME | DESCRIPTION | STARS | OFFICIAL | AUTOMATED |
|--------------------|---|-------|----------|-----------|
| busybox | Busybox base image. | 848 | | [OK] |
| progrium/busybox | | 65 | | [OK] |
| radial/busyboxplus | Full-chain, Internet enabled, busybox made... | 11 | | [OK] |

注：（1）通过 `--filter` 选项筛选，查找更准确。

（2）通过 `--limit` 选项，限制展现数量。

（3）加上 `-no-trunc` 选项，将显示完整信息。

更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/search/>。

7. 上传镜像

可以使用 `docker push` 来上传镜像到仓库，默认上传到 DockerHub 官方仓库（需要登录），也可以上传到第三方的镜像仓库（比如网易云）。

用法如下。

```
Usage: docker push [OPTIONS] NAME[:TAG]
```

举例如下。

```
# 假如你本地有一个 test:latest 镜像
$ docker images
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|--------|--------------|--------------|----------|
| Test | latest | 77af4d6b9913 | 19 hours ago | 1.089 GB |

```
# 添加新的标签 user/test:latest
$ docker tag user/test:latest test:latest
# 推送到指定的镜像仓库（默认是 DockerHub）
$ docker push user/test:latest
```

更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/push/>。

8. 使用 Docker 仓库

正如前文所说，镜像集中存放在仓库（Repository）当中，仓库又可以分为公有仓库和私有仓库。

本章我们将分别展示如何使用 Docker Hub 和网易云镜像中心进行登录、下载等基本操作，最后还将介绍如何创建和使用私有仓库。

注：容易与仓库（Repository）混淆的概念是注册服务器（Registry），实际上注册服务器是存放仓库的具体服务器，每个服务器上可以有多个仓库，而每个仓库下面有多个镜像，每个镜像又可能有不同的标签（tag），和 Git 很相似。举例来说，`hub.c.163.com/public/ubuntu:16.04` 中 `hub.c.163.com` 是注册服务器地址，`public` 是用户工作空间，`ubuntu` 是仓库名，`16.04` 是标签名。

Docker Hub 公有仓库

Docker Hub 是全球最大的 Docker 共有仓库，截止到 2016 年 10 月，Docker Hub 中已经有 40 万个公共镜像（每周 4~5 倍的增长），更关键的是累计有 60 亿次的下载数，可见 Docker Hub 的潜力。

● 登录

通过 `docker login` 来登录，默认登录 Docker Hub。

```
Usage: docker login [OPTIONS] [SERVER]
```

举例如下。

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: bingohuang
Password: xxx
Login Succeeded
```

登录完成之后，将会在用户目录的 `.docker` 目录下，保存用户的认证信息。

● 基本操作

用户可以通过 `docker search` 来查找镜像，通过 `docker pull` 来拉取镜像，再通过 `docker`

push 来推送镜像。

网易云镜像中心

网易云镜像中心分为私有镜像中心和公共镜像中心。

● 私有镜像中心

私有镜像中心是用户私人镜像仓库的管理中心，管理以下两类镜像仓库。

(1) 自定义的镜像仓库：用户有完整操作权限的镜像仓库，镜像来源包括通过 Dockerfile 创建的自定义镜像、从网易云另存为的自定义镜像，以及 Docker 客户端创建的自定义镜像。

(2) 收藏的镜像仓库：从网易云镜像中心 (<https://c.163.com/hub/>) 收藏的官方、第三方公开镜像。

网易云私有镜像中心的层级结构如图 2-4 所示。

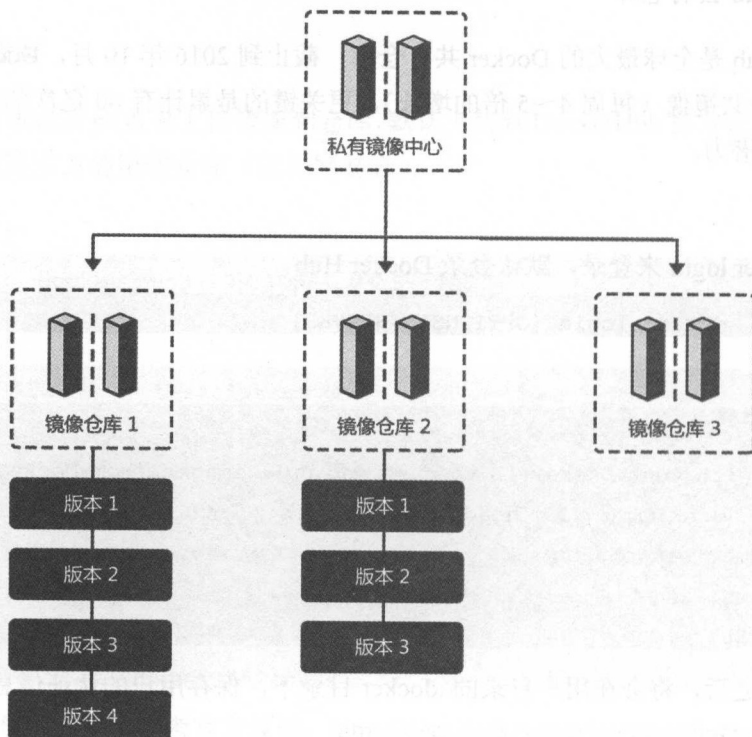


图 2-4 网易云私有镜像中心层级结构

每一层都支持各种操作。

- (1) 私有镜像中心层：支持创建镜像仓库、列出所有镜像仓库等。
- (2) 镜像仓库层：支持通过 Dockerfile 构建新镜像、设置镜像仓库的访问控制权限等。
- (3) 版本层：支持删除一个特定的版本等。

● 公共镜像中心

公共镜像中心为开发者提供了公共镜像仓库服务，无需登录认证就可以下载使用，登录后即可对公开镜像进行收藏和部署，满足开发者的搜索、开发、分享镜像等需求。

使用场景

本地搭建测试环境，可以使用公有镜像中心拉取官方镜像部署。目前，网易云提供以下3种类型镜像。

- (1) Docker Hub 官方镜像。
- (2) 网易云官方镜像。
- (3) 网易云用户公开的镜像。

搭建和使用私有仓库

安装 Docker 后，可以通过官方提供的 registry 镜像来简单快速地搭建一套本地私有仓库环境。

```
$ sudo docker run -d -p 5000:5000 registry
```

默认情况下，仓库被创建在容器的 /tmp/registry 下，可以通过 -v 参数来将镜像文件存放在本地的指定路径。例如下面的例子将上传的镜像放到 /opt/data/registry 目录。

```
$ sudo docker run -d -p 5000:5000 -v /opt/data/registry:/tmp/registry registry
```

私有仓库安装完毕，接下来就可以在私有仓库搜索、下载和上传你的镜像了。

2.2.3 Docker 容器及操作

容器是镜像的运行实例，它可以被启动、开始、停止和删除。每个容器都是相互隔离的、保证安全的独立空间。可以把容器看作是一个简易版的 Linux 环境（包括 root 用户权

限、进程空间、用户空间和网络空间等）和运行在其中的应用程序。

注：镜像是只读的，容器在启动的时候创建一层可写层作为最上层。本节将围绕容器具体介绍其相关的重要操作。

1. 创建容器

Docker 容器十分轻量，可以使用 `docker create` 创建一个容器。

用法如下。

```
Usage: docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

举例如下。

```
$ docker create -t -i fedora bash
6d8af538ec541dd581ebc2a24153a28329acb5268abe5ef868c1fla261221752
$ docker start -a -i 6d8af538ec5
bash-4.2#
```

注：使用 `docker create` 新建的容器处于停止状态，可以使用下面的 `docker start` 来启动它。

```
Usage: docker start [OPTIONS] CONTAINER [CONTAINER...]
```

更详细的使用方式请参考官方说明。

(1) <https://docs.docker.com/engine/reference/commandline/create/>。

(2) <https://docs.docker.com/engine/reference/commandline/start/>。

2. 新建并启动容器

除了上述先创建一个容器再启动，还有一种更简洁的方式，就是 `docker run`，等价于 `docker create + docker start`。

用法如下。

```
Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

举例如下。

```
$ docker run --name test -it debian
root@d6c0fe130dba:/# exit
```

```
exit
$ docker ps -a | grep test
d6c0fe130dba debian:7 "/bin/bash" 26 seconds ago Exited (13) 17 seconds ago
test
```

注：`docker run` 是 Docker 最复杂的命令，选项非常多，以下是 `docker run` 在后台做的操作。

- (1) 检查本地是否存在指定的镜像，不存在就从公有仓库下载。
- (2) 利用镜像创建并启动一个容器。
- (3) 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层。
- (4) 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中。
- (5) 从地址池配置一个 IP 地址给容器。
- (6) 执行用户指定的应用程序。

(7) 执行完毕后容器被终止。如果想让 Docker 容器在后台以守护态 (Daemonized) 形式运行，可以通过 `-d` 参数来实现，如下所示。

```
$ sudo docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello
world; sleep 1; done"
77b2dc01fe0f3f1265df143181e7b9af5e05279a884f4776ee75350ea9d8017a
```

更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/run/>。

3. 查看容器

在上面的示例中，我们通过 `docker ps` 来查看容器，用法如下。

```
Usage: docker ps [OPTIONS]
```

注：(1) 通常，我们会加上 `-a` 选项来查看所有容器的状态。

(2) 通过 `--latest` 或 `-l` 可以查看最新被创建的容器。

更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/ps/>。

4. 进入容器

使用 `-d` 参数，容器启动后会进入后台（或者在容器交互终端按 `CTRL+P` 和 `CTRL+Q` 组合键），用户无法看到容器中的信息。如果需要进入到容器进行操作，一般可以通过 `docker exec`。

用法如下。

```
Usage: docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

举例如下。

```
# 创建一个 ubuntu 容器，并启动 bash 会话
$ docker run --name ubuntu_bash --rm -i -t ubuntu bash
# 在这个容器中创建一个新的 bash 会话
$ docker exec -it ubuntu_bash bash
# 在这个容器中，以后台运行的方式创建一个新文件
$ docker exec -d ubuntu_bash touch /tmp/execWorks
```

更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/exec/>。

5. 终止容器

可以使用 `docker stop` 命令来终止容器。

```
Usage: docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

也可以通过 `docker kill` 命令来终止容器。

```
Usage: docker kill [OPTIONS] CONTAINER [CONTAINER...]
```

区别在于，`stop` 是先发送 `SIGTERM` 信号，一会再发送 `SIGKILL` 信号，而 `kill` 是直接发送 `SIGKILL` 信号。

更详细的使用方式请参考官方说明。

(1) <https://docs.docker.com/engine/reference/commandline/stop/>。

(2) <https://docs.docker.com/engine/reference/commandline/kill/>。

6. 删除容器

可以使用 `docker rm` 命令删除处于终止状态的容器。

| Usage: <code>docker rm</code> | | | [OPTIONS] | CONTAINER | [CONTAINER...] |
|-------------------------------|-----------------|--------------------|-----------|-----------|----------------|
| 举例 | | | | | |
| <code>\$ docker redis</code> | <code>rm</code> | <code>redis</code> | | | |

注：（1）如果要删除一个运行中的容器，可以添加 `-f` 参数。Docker 会发送 `SIGKILL` 信号给容器，终止其中的应用并删除。

（2）如果想要删除所有停止的容器，执行：`$ docker rm $(docker ps -a -q)`。

更详细的使用方式请参考官方说明：<https://docs.docker.com/engine/reference/commandline/rm/>。

2.2.4 基于容器工程化

前面对容器及 Docker 技术进行了简单的介绍，我们大致了解了容器及镜像是什么及如何使用。基于 Docker 进行工程化的核心就是基于 Dockerfile 把原有的构建过程与容器镜像整合，一方面构建流程规范化，另一方面也解决了环境的一致性问题。本节我们将简单介绍结合标准化的构建及 Docker 的镜像进行构建发布的大致流程，下一节我们以实战示例的形式展示怎么整合，体会 Docker 在各个环境下的“构建、发布和运行”。以下 Dockerfile 文件中的步骤体现了大致的流程。

```
FROM maven:latest
RUN apt-get update && apt-get -y install git
# 拉取最新代码
git clone https://github.com/163yun/spring-boot-docker-cloudcomb/tmp/
example
WORKDIR /tmp/example
# 构建代码
RUN mvn -B package
# 代码部署
WORKDIR /
RUN cp /tmp/example/target/spring-boot-docker-cloudcomb-0.1.0.jar app.jar
```

```
# 运行代码
```

```
ENTRYPOINT ["java","-jar","/app.jar"]
```

然后使用此 Dockerfile 来构建镜像 `spring-boot-docker-cloudcomb`。

```
$ docker build -t spring-boot-docker-cloudcomb.
```

构建好镜像后，就可以基于镜像运行服务。

```
$ docker run -it --rm spring-boot-docker-cloudcomb
```

所以，基于容器工程化的核心是基于基础镜像构建自己的应用镜像，一般都需要经过拉取最新代码、构建代码和部署代码的过程。镜像构建完成后，就可以随时基于构建出来的镜像运行服务，因为所有的环境依赖都打包进镜像，不会因为外部环境的差异而导致问题。

2.3 实战示例

工程离不开实践，尤其是软件工程。本节将一步步带你构建一个基于 Spring Boot 和 Docker 的单体应用，通过镜像将其部署在网易云并运行。

第一步：新建工程目录

新建一个文件夹，名字就是你的项目名，这里以 `spring-boot-docker-cloudcomb` 为例。

```
mkdir spring-boot-docker-cloudcomb
```

在根目录下创建 `pom.xml` 文件。

```
touch pom.xml
```

在当前目录下新建子目录。

```
mkdir -p src/main/java/com/bingohuang/hello
```

项目结构如下。

```
spring-boot-docker-cloudcomb
├─ pom.xml
└─ src
   └─ main
```

```
└─ java
    └─ com
        └─ bingohuang
            └─ hello
```

第二步：配置 POM 文件

在 pom.xml 中添加内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>bingohuang.com</groupId>
    <artifactId>spring-boot-docker-cloudcomb</artifactId>
    <version>0.1.0</version>
    <packaging>jar</packaging>
    <name>Spring Boot + Docker + Cloudcomb</name>
    <description>一步步带你构建 Spring Boot + Docker 应用及网易云</description>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.3.RELEASE</version>
        <relativePath/>
    </parent>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
```



```
<scope>test</scope>
</dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>
```

上述 POM 文件并不复杂，是一个 Spring Boot 的标准配置，Spring Boot 的 Maven 插件会提供以下功能。

- 收集的类路径上所有 JAR 文件，并构建成一个单一的、可运行的 JAR 文件，这使得它更便于执行和传输服务。
- 搜索 `public static void main()` 方法来标记为可运行的类。
- 提供了一个内置的依赖解析器，用于设置版本号以匹配 Spring Boot 的依赖。你可以覆盖任何你想要的版本，但它会默认使用 Spring Boot 所设置的版本集。

第三步：编写 Spring Boot 应用

创建一个简单的 Java 应用程序。

```
$ touch src/main/java/com/bingohuang/hello/Application.java
package com.bingohuang.hello;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
/**
 * Spring Boot 主应用入口
 */
@SpringBootApplication @RestController public class Application {
    @RequestMapping("/")    public String home() {
        return "Hello Spring Boot, Docker and CloudComb!";
    }
}
```



```
public static void main(String[] args) {  
    SpringApplication.run(Application.class, args);  
}
```

代码核心就是处理了根路径 / 的 Web 请求，并包含可执行的 main 方法，解释其中几个关键点如下。

- 用 `@SpringBootApplication` 和 `@RestController` 注解类，表示可用 Spring MVC 来处理 Web 请求。
- `@RequestMapping` 将 / 映射到 home 方法，并返回相应文本。
- main 方法使用 Spring Boot 的 `SpringApplication.run` 方法来启动应用。

第四步：本地运行程序

Maven 构建

该应用的核心代码已完成，只有两个文件，可见 Spring Boot 非常简单，目录结构如下。

```
spring-boot-docker-cloudcomb  
├── pom.xml  
└── src  
    ├── main  
    │   ├── java  
    │   │   ├── com  
    │   │   │   ├── binghuang  
    │   │   │   │   ├── hello  
    │   │   │   │   │   └── Application.java
```

在根目录执行如下命令。

```
mvn package
```

之后会在根目录下生成一个 target 目录，并在 target 目录下包含一个可执行的 jar 包。

运行 JAR 包

Spring Boot 的强大之处是将应用打成一个可独立运行的 JAR 文件。

```
java -jar target/spring-boot-docker-cloudcomb-0.1.0.jar
```

不出意外，输出日志，应用启动，默认监听 8080 端口。

访问应用

应用正常启动后，浏览器访问 <http://127.0.0.1:8080/>，即可看到页面输出如下文本。

```
Hello Spring Boot, Docker and CloudComb!
```

第五步：容器化构建及运行

Dockerfile

在项目根目录下创建一个 Dockerfile 文件，内容如下。

```
FROM maven:latest
RUN apt-get update && apt-get -y install git
# 拉取最新代码
git clone https://github.com/163yun/spring-boot-docker-cloudcomb /tmp/
example
WORKDIR /tmp/example
# 构建代码
RUN mvn -B package
# 部署代码
WORKDIR /
RUN cp /tmp/example/target/spring-boot-docker-cloudcomb-0.1.0.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

该 Dockerfile 并不复杂，核心功能就是拉到 Git 仓库中的最新代码并构建，并在容器启动时默认执行启动命令 `java -jar/app.jar`。

此时项目所有源文件编写完成，共 3 个文件，目录结构如下。

```
spring-boot-docker-cloudcomb
├── Dockerfile
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
```

```
└─ bingohuang
   └─ hello
      └─ Application.java
```

构建镜像

在项目根目录下执行镜像构建如下命令。

```
docker build -t spring-boot-docker-cloudcomb:0.1.0.
```

运行容器

```
docker run -p 8080:8080 -t spring-boot-docker-cloudcomb:0.1.0
```

访问项目

同样输出日志，监听 8080 端口，浏览器访问 <http://127.0.0.1:8080/>，输出如下文本。

```
Hello Spring Boot, Docker and CloudComb!
```

第六步：推送镜像到网易云

首先，需要有一个网易云的账号，可在网易云官网 (<https://www.163yun.com/>) 注册。接下来，在命令行中登录云基础服务镜像仓库。

```
docker login hub.c.163.com
Username (test@163.com): test@163.com Password:
Login Succeeded
```

接着，统一标记本地镜像。

```
docker tag spring-boot-docker-cloudcomb:0.1.0
hub.c.163.com/test/spring-boot-docker-cloudcomb:0.1.0
```

最后，推送镜像到云基础服务镜像仓库。

```
docker push
hub.c.163.com/test/spring-boot-docker-cloudcomb:0.1.0
```

第七步：在网易云上创建服务

具体步骤请参考网易云在线帮助文档：<http://support.c.163.com/md.html#!计算服务/容器>

服务/在网易云上创建服务.md。

注：此镜像已经在网易云上公开，地址是 <https://c.163.com/hub#/m/repository/?repoId=41359>，读者也可以直接在云基础服务镜像中心（<https://c.163.com/hub#/m/home/>）搜索 spring-boot-docker-cloudcomb，打开收藏，即可直接基于该镜像创建搜索 Spring Boot + Docker 的应用服务。

参考网址

- 源代码：<https://github.com/163yun/spring-boot-docker-cloudcomb>
- Spring Boot：<http://projects.spring.io/spring-boot/>
- Docker：<https://docker.com>
- 网易云：<https://www.163yun.com>

小 结

本章首先对实际项目中的一些工程化实践经验进行了介绍，包含工程模板、模板化、工程化构建、代码版本管理及环境划分，并对在实际发布过程中的环境一致性问题进行了描述，然后基于容器技术来解决环境一致性问题，并对容器相关技术进行了简单介绍，最后以示例形式演示了如何使用容器技术整体构建工程。总之，本章的目的是帮助读者了解一些工程化的实践经验及容器技术，后续才能以此为基础快速进行产品的开发及发布。

第3章 初创期应用架构实践

当我们有了一个应用构想的时候，往往需要快速实现一个原型进行验证及试错。在开始阶段，应用的业务方向调整比较频繁，此时，我们更多是考虑一个应用基本功能的实现，以及能否快速迭代以跟上业务方向的调整，实现一个应用的快速成长，并且在实践过程中尽可能做到为后续的架构升级做好铺垫。

在应用的初创阶段，访问量并不大，一个比较基础的单体架构的系统足以支撑起整个业务。因此，本章我们不涉及服务拆分、服务化这些内容，而是从做出一个完整的系统出发，主要的任务是完成基本的业务需求，这些需求包括：选择合适的技术栈；能够快速进行功能的迭代开发，做好基本的高可用；规范整个开发流程；从一开始就要考虑到应用安全，不犯常见的安全错误；应用部署后，对应用进行基础的监控，更早地发现系统缺陷及性能问题。因为随着业务的成型，访问量随时可能暴涨，对应用监控的主要目的是保证我们的产品能够正常地服务用户，及时发现应用故障，以期快速解决故障。

3.1 技术选型

一个基本的单体架构，其功能是以实现产品业务为准，一般来说，其功能需求主要包括业务逻辑如何实现及相关的数据如何存储。业务逻辑的实现一般涉及相关业务技术的选型，即如何选择合适的框架来支撑业务。关于数据存储，针对不同的数据类型可以选择不同的存储方式：对于结构化数据，一般选择关系型数据库；对于非结构化数据，一般选择键值对数据库；如果有大量的图片、视频的存储需求，一般选择专门的对象存储系统进行存储。

3.1.1 业务框架选型

当我们实现一个应用的时候，最先面对的就是使用什么样的应用框架。一个强大的应

用开发框架可以大大节省我们的时间和精力，避免重复造轮子。

1. 选型因素

在做具体的业务框架选型的时候，往往要考虑常见的选型因素，以指导我们做出比较合理的选型。

业务相关性

业务因素是我们在做框架选型时需要首先考虑的因素。当我们实现一个特定的业务功能时，某些编程语言中恰好有对应的三方库实现，这个时候选择这门编程语言，往往能达到事半功倍的效果。当我们选择好一个框架后，也就选择了一门编程语言，因此该编程语言下的三方库是否足够多，一些常见的功能是不是都能实现也是一个考虑因素。如果我们的业务更多是数据库的增删查改，那么当框架提供了接口良好的数据库抽象，有比较好的数据持久化支持时，就会大大提高我们的开发效率。如果我们的业务是一个基于长连接的实时服务，那么使用纯异步的 Node.js 可能更合适。因此，框架的选型与我们的业务类型息息相关，也是我们首要考虑的因素。

框架流行度

越流行的框架，使用的人越多。使用的人越多就意味着，当我们在遇到问题的时候，就会有更多潜在的人会回答我们的问题。一个活跃的社区，可以帮我们解决绝大部分问题，因为很多问题在我们遇到之前，已经有别人遇到了。当我们在网上搜索相关问题的时候，也许可以直接定位到答案。

越流行的框架，学习的人就越多。当我们需要招聘新人的时候，也更容易招到对口的人，减少我们的招聘成本。因此在做框架选型的时候，不要为了图“新奇”，图“品味”，去选择一些比较小众的框架或者小众的编程语言。

这里还要说明的一点就是，流行度是有时间因素的，前些年比较流行的框架，到了现在可能已经不再流行了，比如 Struts 2。而当今的一些流行框架，随着时间的推移，也可能被更新的框架所取代。因此这里的流行度，都是相对在一定时间内的。

学习曲线

每一个框架都有自己的特性，命名规范、文件夹结构和使用方式都不相同。有些框架

可以很灵活地配置，但是带来了配置的复杂性；有些框架则使用“约定优先配置”的思想对配置进行了约定，使得配置非常简单，但是面对复杂业务的时候又表现得力不从心。

这些不同带来了面对不同业务时候的不同学习曲线。陡峭的学习曲线，往往为学习者带来太多学习的负担。而一个学习简单、配置简单的框架，则可以减少后续团队增加新成员时的成本，并使运维配置也变得简单。

还有，如果对这个框架所使用的语言不熟悉，同时学习语言加框架会导致学习曲线更加陡峭，选择一个用熟悉的编程语言写的框架往往学习起来更轻松。因此在选择框架的时候，尽量选择比较流行的编程语言写的框架。

更进一步地说，框架源码如果有比较好的可读性，当我们需要了解这个框架的实现细节时，也能比较快速地入门去阅读源码。因此，框架源码可读性的好坏，在一定程度上，也影响了我们对这个框架具体实现细节的了解，从而影响我们对这个框架的深入应用。因此，请尽可能选择源码可读性好的框架，使得源码阅读的学习曲线比较平缓。

文档

一个具有良好文档的框架，往往是我们使用这个框架的保证。框架中大量的代码片段、例子代码和教程都可以帮助我们更好地使用这个框架。而一个文档匮乏的框架，无论吹嘘得有多好，没有良好的文档，我们将不得不去摸索使用，甚至不得不去阅读其代码。这些都会减缓开发的效率。不过很幸运的是，很多优秀的框架，也都有着优秀的文档。

社区支持

如果一个框架非常适合你的业务逻辑，但是这个框架已经很久没有更新了，那么在做框架选型的时候，就需要特别小心。一个长期不更新的框架，或许已经被社区放弃了。使用该框架后，一旦遇到了问题，我们无法从官方社区获得支持，只能靠我们自己去解决，可能会拖慢我们的开发进度。

单元测试

在开发过程中，单元测试是必不可少的部分。单元测试能够尽快发现某一部分代码的bug，提高代码质量，减少项目集成的成本。因此，如果框架提供了对单元测试的支持，用户可以在框架之上直接写自己的单元测试逻辑，以提高开发效率。

可扩展性

框架是否提供了扩展接口也是我们要考虑的因素。随着业务的发展，框架的基本功能逐渐不再满足我们的功能及非功能需求。如果其提供了良好的扩展接口，我们就可以基于这些扩展接口，实现自己的一些定制功能。

许可证

虽然目前绝大部分应用框架都是开源的，但是一些框架可能还需要付费使用，或者一个框架的某些扩展部分需要付费使用。因此在做框架选型之前有必要去简单了解一下这个框架的授权协议，以防在使用中出现知识产权的问题。

上面这些选型因素之间不是孤立的，他们是相辅相成的。一个流行的应用框架，一般都具有良好的文档，良好的社区支持，可扩展性很强，适合绝大部分场景下的业务，学习入门简单等特点。反之一个学习入门简单，文档齐全的框架又会有更多的人去使用，形成一个良好的社区文化，使得框架更流行，从而又会反哺框架的发展，形成一个良性循环。

在应用的初始阶段，我们可以不断地通过原型试错，从而选择一个比较适合我们具体业务的框架，做一个比较好的技术选型。而一旦选定了技术框架，随着我们在框架上面开发的业务越来越多，后续如果想做框架转换将会越来越困难。因此在最初的框架选型阶段有必要非常慎重。

另外，当我们为了完成某个特定的业务功能时，可以使用的三方库的选择也有很多。一个合适的三方库的选择同样需要考虑上面讨论到的因素，文档是不是完整，许可证是不是友好，用的人是不是很多等。一个好的选择，会带来更少的问题，提高开发效率，使得我们能够更快地进行迭代。

2. Java 语言的框架选型

下面我们选择最普遍的一种业务类型，也就是使用 Java 语言做 Web 开发，来说明如何进行框架选型。

控制反转及依赖注入

Java 是一门面向对象的编程语言，我们在应用系统设计的时候，往往也会采用面向对象的方式来设计。在使用面向对象的方式来设计系统实现的时候，我们面对的业务逻辑，

往往涉及两个或是更多的类通过彼此的合作来实现，这使得每个对象都需要获取或者实例化与其合作的对象。如果这个过程要靠对象自身主动装配的话，就会导致代码高度耦合并且难以维护和调试。当我们更新了某一个模块的实现时，即使其接口没有任何变化，由于是代码中应用主动实例化或者主动去获取的，我们将不得不修改相应的代码，还需要对项目重新编译，浪费大量的精力和时间。

为了解决这样的问题，控制反转的概念被提出。一个支持控制反转的框架容器，可以根据用户配置，去完成相关对象的实例化，并且装配好对象间的依赖，完成依赖注入。当实现改动的时候，我们只需要修改一下配置即可实现新版本的替换，而不需要在代码中主动去初始化对象，主动去装配对象的依赖。使用控制反转的设计方法，可以使代码间的耦合延迟到运行时进行，通过配置的方式，由相应的框架来完成对象的创建及装配。

目前已有的控制反转的容器有 PicoContainer、Apache Excalibur、Spring IoC 等，其中最流行的也是名气最大的是 Spring IoC。Spring IoC 提供了基于 Annotation 及 XML 的两种配置方式，用户可以根据实际情况具体选择。总的来说，使用 Annotation 的方式，比较简单，因为 Annotation 是写在代码中的，与代码有耦合；使用 XML 的方式，相对烦琐，因为完全与代码无耦合。Spring IoC 使用反射机制，支持多种方式的对象的创建及依赖注入，可以满足我们的绝大部分需求。Spring IoC 作为 Spring 框架的一部分，文档丰富，官方维护活跃，相比于其他的控制反转框架，Spring IoC 是一个不错的技术选择。

MVC 模式

作为最普遍的一个 Web 应用，最常见的一种软件架构模式是 MVC 模式。MVC 模式把一个系统分为 3 部分：模型（Model）、视图（View）和控制器（Controller）。视图就是直接面向最终用户的界面，对于一个 Web 应用来说，可以认为就是展示给用户的网页渲染。模型是程序员编写程序应完成核心功能的地方，如果其主要功能就是进行增删查改的话，那么模型可以认为主要与数据库打交道、负责数据访问，以及完成对数据库的相关操作。控制器的主要作用是控制应用的行为，负责接收派发请求，与模型交互，然后将相关的结果投递给视图，由视图来决定怎么向用户展示，视图往往还会接受来自用户的操作。MVC 之间的关系如图 3-1 所示。

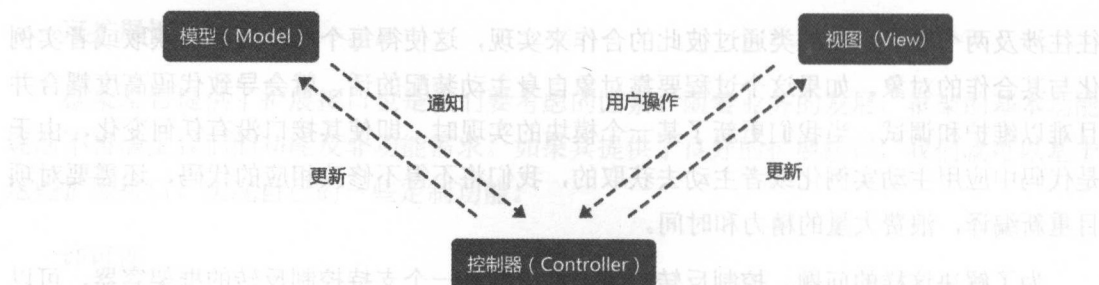


图 3-1 MVC 之间的关系

在 MVC 模型中，模型不关心具体的数据是如何被展示以及如何被操作的；控制器不关心模型中的数据是从哪里来的，也不关心视图到底是怎么展示数据的；视图仅仅关心数据如何被呈现，并不关心这些数据是如何通过逻辑处理获取的。

使用这种分层模型，层与层的职责分明，当我们有需求变动时，可以快速响应。比如，数据库要进行替换，从 Oracle 迁移到 MySQL，我们仅仅需要改动模型层的相关接口的实现代码即可，而不会影响到视图与控制器。当我们的应用需要更友好地向用户展示数据的时候，我们仅仅需要更新视图代码，对于 Web 应用来说，一般是前端页面的代码，不会影响到模型及控制器。当我们需要对一些请求处理逻辑进行调整的时候，仅仅需要调整控制器即可，不会影响到具体的模型及视图。对于应用开发来说，不同的模块可以独立开发，在后续的可扩展性，可维护性方面都好很多。由于各个模块代码是解耦的，也提高了代码的可重用性，比如同样的一个视图可以用于不同的应用，无非是渲染的数据有不同，同样的一个模型中的数据操作也可用于不同的应用。

在 Java 语言中，常见的 MVC 框架有 Struts 2、Spring MVC 等。Spring MVC 是目前很流行的 Web MVC 框架，该框架上手简单、文档齐全、社区活跃，很多学校及培训机构都有 Spring MVC 的课程，因此可以降低招聘成本，容易招到合适的人。同时，它还可以与 Spring 框架无缝集成，Spring 作为一个企业级应用开发框架，提供了包括控制反转容器等组件，使用这些组件，能够提高我们的开发效率，避免重造轮子。

相比于 Struts 2，Spring MVC 的基于方法设计实现的控制器，比起 Struts 2 基于 Action 类的实现更灵活，学习入门也更直观简单。由于不用每次都去实例化一个类，因此性能上也会稍好。Spring MVC 对 URL 映射处理也更优雅，可以更好地支持 RESTful 风格的 API。再加上近年来，Struts 2 出现了几次安全漏洞，因此，对于 MVC 框架选型来说，Spring MVC 是一个不错的选择。

数据库持久化框架

在 MVC 模式中，很多场景下，模型的具体业务逻辑都是对数据库的增删改查。由于 Java 是一门面向对象的语言，数据库往往都是关系型数据库，我们需要一个框架将数据库中的数据与业务对象之间做一个关联，当我们需要访问数据库的时候，只需要与映射过的业务对象打交道就好了。否则，如果每次数据库访问都是手动写 SQL 语句，通过 JDBC 执行，我们的代码里面就会有大量的 SQL 拼接代码，给维护带来困难。

在 Java 语言中，Hibernate 是一个流行的数据库持久化框架，它能够自动生成并执行 SQL 语句，完成对数据库中的记录到业务对象的自动映射（对象关系映射，ORM），将对数据库的修改操作转换成对业务对象的操作，数据库无关性好。比如，通过修改一个自动映射业务对象的某一个属性的值，就可以自动完成与其对应的数据库中记录的修改。通过这种面向对象的编程方式去操作数据库，给我们的编程带来了便利，也避免了代码中拼接 SQL 语句。

在使用 Hibernate 的时候，由于 SQL 语句往往都是自动生成并执行的，这些自动生成的 SQL 语句并没有很好地被优化，对于一些复杂查询感到吃力。另外，由于性能的原因，很多时候，我们需要对 SQL 语句进行优化。相比于 Hibernate，另一个流行的数据库持久框架 Mybatis 着重于业务对象与 SQL 语句之间的映射关系，用户可以手动书写 SQL 语句，对 SQL 语句进行优化，并通过配置文件来指定 SQL 语句的参数如何映射，返回的结果集到业务对象如何映射，Mybatis 会根据用户的配置来完成 SQL 语句参数的映射及对返回结果集的处理。

在实际应用中，刚开始应用业务较简单，使用 Hibernate 有很高的开发效率，因为可以完全不关心 SQL 语句，对数据库的所有操作都映射成了对业务对象的操作。但是随着业务的发展，所需要的查询会越来越复杂，此时自动生成的 SQL 往往不再满足具体的查询需求。而使用 Mybatis 的话，由于可以手动写 SQL，对 SQL 优化，在面对复杂查询方面会更为得力。Mybatis 这种“半自动化”的特性带来了极大的灵活性，因此，考虑到后续可能的复杂查询需求，相比于 Hibernate，使用 Mybatis 是一个不错的选择。当然，这个选型不是绝对的，我们也可以根据不同的场景，混合使用这两个框架，来应对不同类型的查询需求。

运行时的 JVM 参数配置

使用 Java 完成应用程序开发后，在运行时，通常需要根据实际业务情况去调整 JVM 的一些参数配置。最常见的就是要调整运行时的堆内存大小参数。其单位一般配置使用 M 或 G，对于 Tomcat 来说，Tomcat 使用一个环境变量 `JAVA_OPTS` 来实现 JVM 配置的注入，因此我们只需要在启动容器的时候，定义一下这个环境变量即可，比如可以将 `JAVA_OPTS` 这个环境变量的值定义为“-Xms512m -Xmx512m”，这样就定义了 JVM 在运行时初始堆大小与最大堆大小都是 512m。对于服务端应用来说，我们一般把-Xms 跟-Xmx 设为一样大，防止运行过程中动态调整堆内存。

3.1.2 结构化数据存储

数据是互联网业务最核心的价值，管理数据是互联网应用最重要的功能。在电商业务场景中，一次会员购买商品的行为，首先涉及商品数据、会员数据的查询，会员点击购买后，生成订单数据，会员支付成功后，又会涉及订单数据的更新。要实现高效的数据管理，数据存储是关键，因为数据的存储方案决定了数据的访问方式及访问效率。

电商业务涉及非常多的数据，按照数据结构，我们可以分为结构化数据和非结构化数据。在电商业务场景中，每一个商品都拥有编号、名称、品牌、货源地、分类等固定的属性，每个属性都有明确的数据类型和长度约束，例如商品名称是字符类型的，有最长字符限制，商品编号是数字类型的，有固定的取值范围，我们将这类有固定结构的数据称为结构化数据。与结构化数据相对应的是非结构化数据，例如商品图片、广告视频，这类数据没有固定的结构。不同的数据类型，数据的存储方案也不相同，我们接下来聊聊结构化数据的存储解决方案。

1. 关系型数据库

关系型数据库是目前使用最为广泛的一种结构化数据存储解决方案，在关系型数据库中，数据被组织成由行和列组成的二维表结构逻辑实现。关系型数据库最早可以追溯到 1970 年，由英国科学家埃德加·弗兰克·科德提出，经过近半个世纪的发展，这个领域涌现出非常多优秀的产品。下面根据图 3-2 所示的 DB-Engines 发布的数据库排名，介绍在互联网业务中应用最为广泛的几个数据库产品。

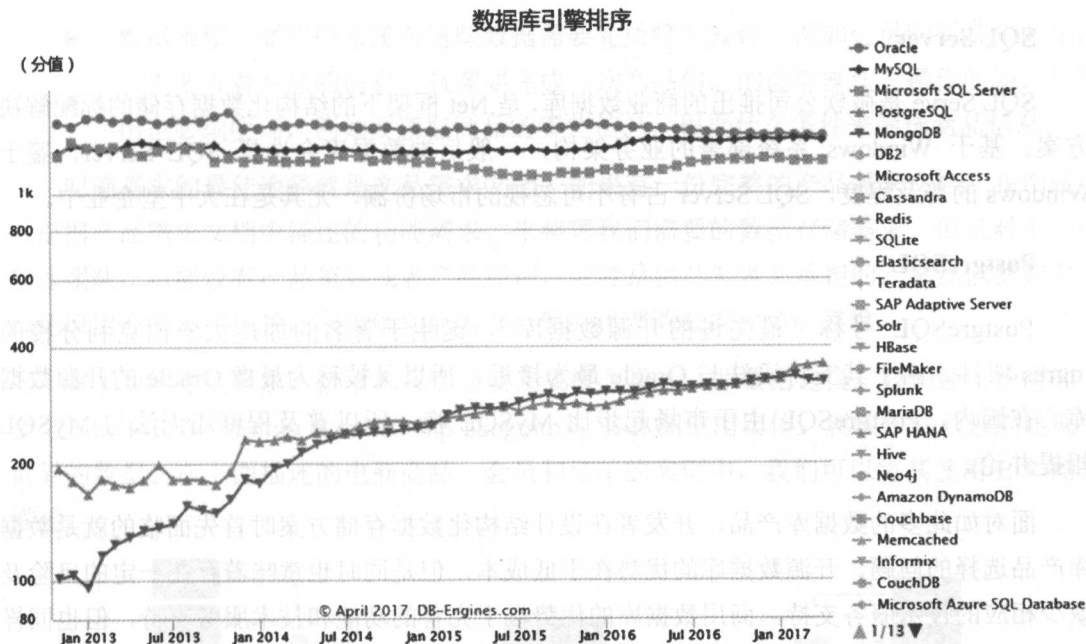


图 3-2 DB-Engines 发布的数据库排名

Oracle

Oracle 是由甲骨文公司研发的一款关系型数据库管理系统，堪称史上最成功的商业数据库，在服务高可用、数据高可靠、查询语法支持、性能、运维工具等方面均有成熟完善的解决方案，同时提供了丰富的监控诊断信息，方便数据库管理员或者开发者进行性能调优。但是价格不菲的配套硬件及许可证授权费用，让很多中小型企业望而却步，目前，Oracle 数据库主要应用于大型企业，在金融、银行业务领域应用较为广泛。但是随着以 MySQL 为代表的开源数据库快速兴起，越来越多的互联网公司开始将业务迁移到开源数据库中，并兴起去“IOE”（O 特指 Oracle）的浪潮。

MySQL

MySQL 是业界最为流行的开源数据库，由芬兰人 Ulf Michael Widenius (Monty) 编写，依靠强大的社区支持及开源浪潮，迅速席卷了各大互联网公司。它具有部署简单、开源免费、使用门槛低、日趋完善的功能特性，以及基于复制可以快速实现一套高可用架构等诸多优势，因此成为中小型企业和创业公司的首选。

SQL Server

SQL Serve 是微软公司推出的商业数据库,是.Net 框架下的结构化数据存储的标配解决方案。基于 Windows 系统部署的业务架构,一般后端数据库会选择 SQL Server,鉴于 Windows 的普及程度,SQL Server 占有不可忽视的市场份额,尤其是在大中型企业中。

PostgreSQL

PostgreSQL 号称“最先进的开源数据库”,诞生于著名的加州大学伯克利分校的 Ingres 项目。由于其支持语法与 Oracle 最为接近,所以又被称为最像 Oracle 的开源数据库。在国内,PostgreSQL 由于市场起步比 MySQL 晚,所以普及程度还无法与 MySQL 相提并论。

面对如此多的数据库产品,开发者在设计结构化数据存储方案时首先面临的的就是数据库产品选择的问题。开源数据库的优势在于低成本,但是同时也意味着存在一定的风险及缺少相应的技术服务支持。商用数据库的优势在于完善的功能和技术服务支持,但也同样价格不菲。随着开源数据库功能的日趋完善和强大的社区支持,越来越多的人才投入到开源数据库提供技术服务支持,开源数据库成为开发者的首选数据库。

2. 云环境下的关系型数据库

目前几乎所有的公有云服务都提供了 MySQL 数据库的云服务,网易云基础服务也不例外,不仅提供了数据库的快捷部署及高可用架构,还提供了技术和服务支持,让开发者即使使用开源数据库,也能获得高质量的服务保障。

需求建模

选定数据库产品后,进入数据库设计阶段,需求建模是数据库设计的第一步,它包括需求分析和概念设计两个步骤。需求分析就是要明确系统的数据存储需求,主要包括以下几个方面。

- 存储内容需求:要明确哪些数据是需要存储的,在前面介绍的电商业务案例中,商品信息、订单信息及会员信息都是我们系统需要存储的数据。
- 数据限制与约束:要明确存储的数据类型、长度限制及数据的约束,包括是否允许为空值,是否必须全局唯一等。

● 数据操作：要明确系统对这些数据需要完成哪些操作，例如会员购买某个商品，首先要查看商品的信息，就需要完成一次商品信息的读取操作，操作的输入与输出也必须明确，例如要读取商品的哪些信息，根据什么条件来筛选商品信息。

明确需求的最佳途径就是产品需求文档，如果有一份完整的产品需求文档，我们就可以根据产品需求文档中描述的功能需求，来整理我们需要的数据存储需求。但是对于一些创业团队，可能没有产品策划或者产品经理，或者是由开发者来承担的，那我们就需要通过还原用户的使用场景，来明确功能需求，然后再理清数据存储的需求。

有了数据存储需求，接下来我们就要完成概念设计，即数据建模。我们经常使用的一个工具即 E-R 图（Entity Relationship Diagram）。E-R 图使用实体、属性和联系来描述现实世界的数据库。在上面描述的电商商品、会员和订单的案例中，我们可以将其使用 E-R 图来描述。

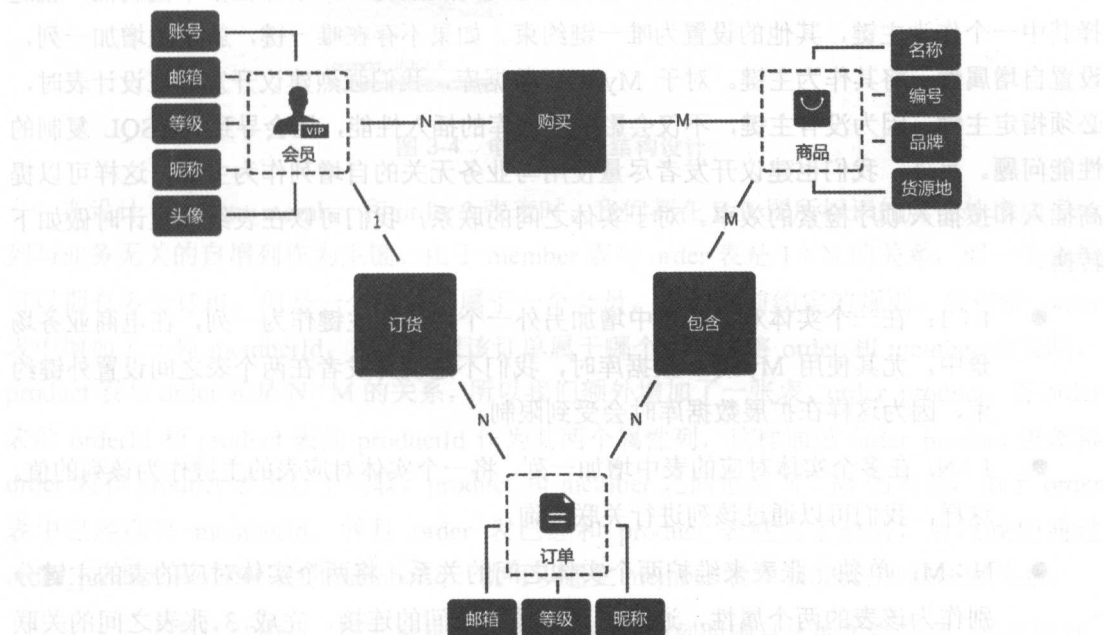


图 3-3 电商系统 E-R 图

图 3-3 中，会员、商品和订单是 3 个实体，我们使用虚线框来表示，3 个实体分别有一些属性来描述这些实体的特征，我们使用矩形框来描述。3 个实体之间是有联系的，我们将实体之间的关系使用实心正方形框来描述，会员和商品之间存在“购买”联系，一

个会员可以购买多个商品，一个商品也可以被多个会员购买。会员和订单之间存在“订货”联系，会员下订单以后，会生成订单数据，一个会员可以产生多笔订单，但是一个订单，只能属于一个会员。订单和商品之间存在“包含”联系，一个订单包含多个商品，同时，一个商品也可以被多个订单包含。在菱形框的两侧我们用连接线来连接有联系的两个实体，并且在连接线上用 1:1、1:N、N:M 来分别表示实体之间 1 对 1、1 对多和多对多的对应关系。目前，很多画图软件都集成了 E-R 图模板，比如 Visio，开发者可以根据场景选择使用。

表结构设计

完成 E-R 图后，我们就可以开始将 E-R 图转换成关系型数据库中的表结构。一般来说，我们将 E-R 图中的实体单独成为关系型数据库中的一张表，将实体的属性定义为每张表的列，如果属性中有能够唯一标识的键，就将其定义为表主键，如果存在多个键属性，就选择其中一个作为主键，其他的设置为唯一键约束。如果不存在唯一键，就单独增加一列，设置自增属性，将其作为主键。对于 MySQL 数据库，我们强烈建议开发者在设计表时，必须指定主键，因为没有主键，不仅会影响数据库的插入性能，还会导致 MySQL 复制的性能问题。另外，我们也建议开发者尽量使用与业务无关的自增列作为主键，这样可以提高插入和按插入顺序检索的效率。对于实体之间的联系，我们可以在表结构设计时做如下转换。

- 1:1: 在一个实体对应的表中增加另外一个实体的主键作为一列，在电商业务场景中，尤其使用 MySQL 数据库时，我们不建议开发者在两个表之间设置外键约束，因为这样在扩展数据库时会受到限制。
- 1:N: 在多个实体对应的表中增加一列，将一个实体对应表的主键作为该列的值。这样，我们可以通过该列进行关联查询。
- N:M: 单独一张表来维护两个实体之间的关系，将两个实体对应的表的主键分别作为该表的两个属性，通过这两个属性之间的连接，完成 3 张表之间的关联查询。

经过转换，电商案例中的 3 张表，设计如图 3-4 所示。

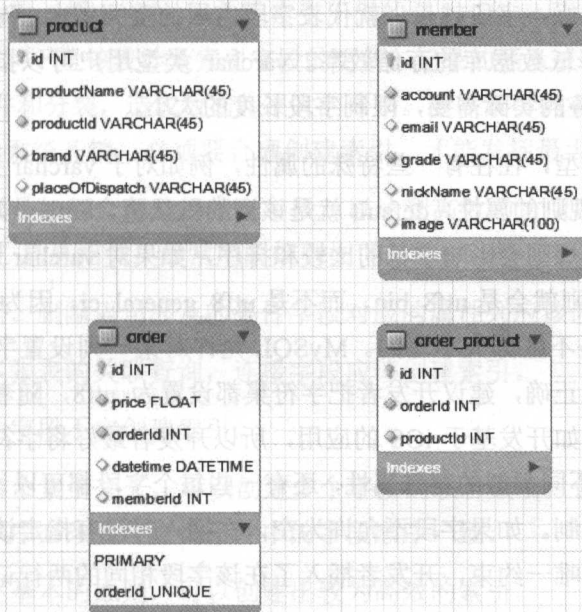


图 3-4 电商系统表结构设计

在设计 **product**、**member** 和 **order** 3 张表时，我们都在 E-R 图所标识的属性外增加了一列与业务无关的自增列作为主键。由于 **member** 表与 **order** 表是 1 : N 的关系，即一个会员可以拥有多个订单，但是一个订单只属于一个会员，根据之前约定的规则，我们在 **order** 表中增加了一列 `memberId`，用来标识该订单属于哪个用户，将 **order** 和 **member** 表关联。**product** 表与 **order** 表是 N : M 的关系，所以我们额外增加了一张表，**order_product**，将 **order** 表的 `orderId` 和 **product** 表的 `productId` 作为其两个属性列，这样通过 **order_product** 表就将 **order** 表和 **product** 表进行了关联。**product** 和 **member** 之间也是 N : M 的关系，由于 **order** 表中已经拥有 `memberId`，并且 **order** 表已经和 **product** 表建立了映射，所以我们通过 **order_product** 和 **order** 表的 `memberId`，就可以建立 **member** 表和 **product** 表的映射关系。

定义了表整体结构后，接下来我们就要针对每一个列明确具体的数据类型、长度限制、约束及索引相关内容。MySQL 数据库提供了丰富的数据类型，常用的包括 `int`、`long`、`float`、`double`、`varchar`、`datetime` 等，根据字段表达的内容确定某一列的数据类型并不困难，但是值得注意的是每一种类型的长度限制，例如 `int` 类型的取值范围是 0~65535，如果超出该值，数据库就无法正确存储该值，如果字段可能大于 65535，就应该使用 `long` 类型。但并

不是字段长度越长就越好，字段越长，就代表字段占用的空间越大，对于数据库来说，会耗费很多存储空间，降低数据库的存储效率。`varchar` 类型用户可以指定字段长度为 `0~65535`，开发者根据业务的实际需要，限制字段长度的大小。

针对不同的数据类型，往往有一些特殊的属性，例如对于 `varchar` 类型，会有 `default`、`binary`、字符集及校验规则的属性，`default` 就是该列的默认值，`binary` 实际与该列的字符校验规则有关，字符校验规则会影响字符的比较和排序，如果对 `varchar` 设置了 `binary` 属性，字符集是 `utf8`，校验规则就会是 `utf8_bin`，而不是 `utf8_general_ci`，因为 `utf_bin` 是区分大小写的，`utf_general_ci` 是不区分大小写的。MySQL 允许对某一列设置字符集，在开发应用时，为了确保中文输入正确，建议开发者把字符集都设置为 `utf8`，随着很多存储的数据会包含一些表情符号，比如开发基于 `iOS` 的应用，所以开发者最好将字符集设置为支持表情符号的 `utf8mb4`。除了不同类型的特殊属性，还有一些每个字段都可以设置的约束限制，有是否允许为空、唯一限制。如果字段不允许为空，在插入时没有指定该列的值，数据库就会抛错误。如果限制了唯一约束，开发者插入了在该字段相同的两行记录，后插入的也会抛错。

最后，我们需要完成索引的设计，这也是表结构设计中难度最大，又最为关键的一部分内容，它对后续数据库访问性能有非常大的影响。我们需要了解 3 个问题。

- 为什么要创建索引？
- 创建索引会对数据库产生什么影响？
- 如何选择索引字段？

第一个问题，在 MySQL 数据库中，存在两类索引，分别是主键索引和二级索引。在默认的 InnoDB 存储引擎中，数据记录是按照主键字段构建的 B+树的组织存储的，如果开发者在创建表时，没有指定主键，InnoDB 也会默认为该表自动生成一个主键，但是我们非常不建议开发者这么使用，因为这会造成一系列性能问题。所以主键是数据存储结构构建的依据，每张表都要有。第二种就是二级索引，它的存在主要是基于性能的考虑，如果在某个字段或者某几个字段创建一个二级索引，则基于这些字段的查询会相对更快，代价更低。

第二个问题，既然二级索引能够加速数据库的访问，是不是创建得越多越好呢，这与索引在数据库的内部实现相关。实际上数据库创建索引是有开销的，创建一个二级索引，

数据库系统就会新构建一棵 B+树，与主键索引不同的是，二级索引的叶子节点不是数据库记录，而是主键的值。一旦该表涉及索引字段的更新，记录的插入或者删除，就会导致二级索引 B+树的节点合并和分裂，造成数据库很大的开销。所以索引确实可以加速数据库访问，但是也会造成一定的系统开销，必须要合理创建索引，才能发挥最大的功效。

最后一个问题就是如何合理选择索引字段，以达到高效的访问。索引字段的选择与应用访问数据库的 SQL 息息相关。最常用的规则如下。

- 查询、更新、删除语句涉及的条件字段对应的属性列应该创建索引。
- 如果存在多张表的关联查询，连接字段应该创建索引。
- 频繁更新的字段不宜创建索引。
- 如果存在多个索引交叉，可以创建多个字段的联合索引，但是需要注意的是，索引的字段顺序必须要与查询条件中的字段顺序一致。

开发者遵循上述基本的原则，可以创建出较为高效的索引。

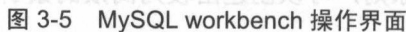
设计工具

目前业界已经有很多优秀的表结构设计工具，帮助开发者完成表结构的设计，并可以直接转化成对应的 SQL 语句在指定的数据库上实施，降低了开发者编写 SQL 语句和执行的烦琐过程。

MySQL workbench 是一款专门针对 MySQL 打造的数据库设计和建模工具，集成了 SQL 开发、数据库管理、数据库设计、创建及维护于一体的客户端工具。其操作界面如图 3-5 所示。

Navicat 也是国内开发者使用较多的一款图形化数据库管理客户端软件，它不仅提供了面向 MySQL，还包括 Oracle、SQL Server 等多种数据库，它是一款收费的软件。其操作界面如图 3-6 所示。

phpMyAdmin 是一款免费开源的基于 Web 浏览器的可视化 MySQL 和 MariaDB 管理系统，它使用 PHP 语言开发。相比于客户端工具，phpMyAdmin 具有免安装的优势，可以随时随地完成对数据库的设计和管理。其操作界面如图 3-7 所示。



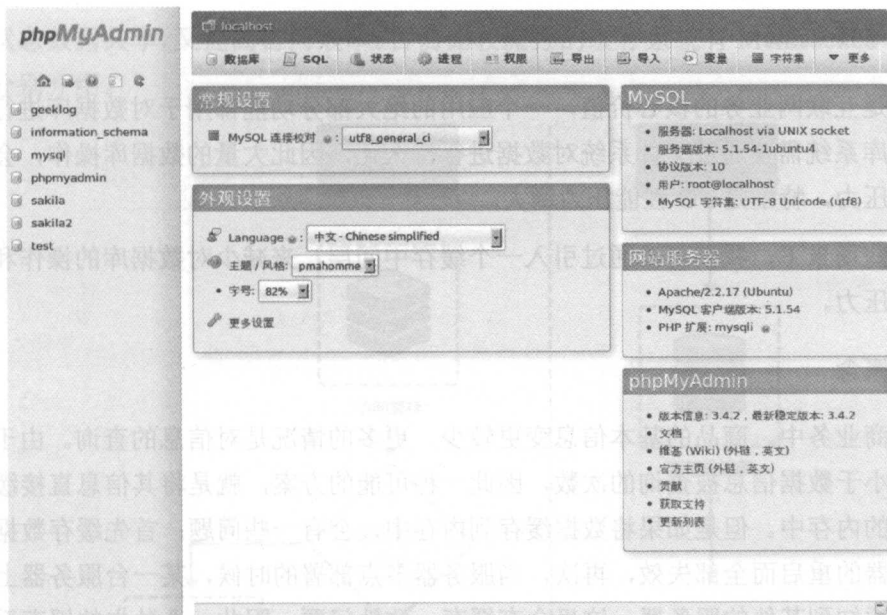


图 3-7 phpMyAdmin 操作界面

公有云数据库服务也面向开发者提供了数据库设计和管理的可视化工具，例如在网易云上的 WebSQL 功能，在网易云数据库的控制台上就可以轻松完成数据库的设计工作。其操作界面如图 3-8 所示。

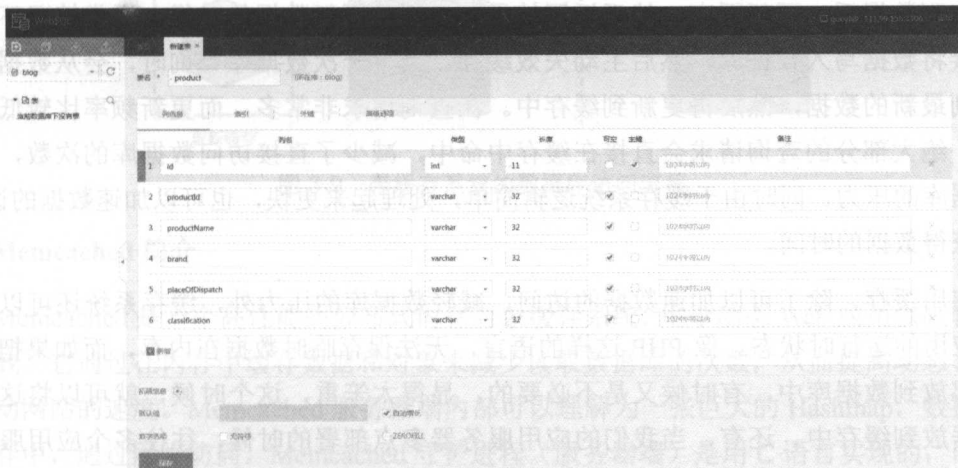


图 3-8 网易云 WebSQL 操作界面

3.1.3 缓存选型

数据是互联网业务的核心价值，一个应用的绝大部分功能都用于对数据库进行增删改查。数据库系统需要通过文件系统对数据进行持久化，因此大量的数据库操作，会使数据库的性能压力，特别是 I/O 性能压力增大。

在某些场景下，我们可以通过引入一个缓存中间层，来减少对数据库的操作和对数据库的访问压力。

缓存概念

在电商业务中，商品的基本信息变更较少，更多的情况是对信息的查询。由于其数据变更远远小于数据信息被查询的次数，因此一种可能的方案，就是将其信息直接缓存到应用服务器的内存中。但是如果将数据缓存到内存中，会有一些问题：首先缓存数据会随着应用服务器的重启而全部失效，再次，当服务器多点部署的时候，某一台服务器上的更新操作需要通知到其他的服务器，这里会有缓存一致性问题，因此一个独立的缓存系统是必要的。目前常用的缓存系统，一般都支持 KV 键值对型，使用简单方便，易于理解。在使用缓存系统的情况下，对数据库的访问流程通常如图 3-9 所示。

当需要获取数据的时候，会先访问缓存，如果在缓存中查询到，则直接返回，省去了对数据库的读查询开销。如果在缓存中没有查询到，则还是需要访问具体的数据库，在获取到数据后，更新缓存，然后返回结果。当需要更新数据的时候，通常情况下，可以直接将数据写入数据库，然后主动失效缓存。当下一次数据库查询时，会从数据库中获取到最新的数据，然后再更新到缓存中。在查询请求非常多，而更新频率比较低的场景下，绝大部分的查询请求会直接在缓存中命中，减少了直接访问数据库的次数，减轻了数据库的压力。同时由于缓存系统逻辑简单，处理起来更快，也可以加速数据的读取，减少获得数据的时间。

使用缓存，除了可以加速数据的访问，减轻数据库的压力外，缓存系统还可以用来保存应用的运行时状态。像 PHP 这样的语言，无法保存临时数据在内存，而如果把所有数据都放到数据库中，有时候又是不必要的，显得太笨重，这个时候，就可以将这些临时数据放到缓存中。还有，当我们的应用服务器多点部署的时候，往往多个应用服务部署实例要共享一些运行时的临时状态数据，比如 Session 信息。将这类数据放到数据库中，既没必要，又会影响访问性能，我们同样可以将这样的状态数据放到统一的缓存中，既

实现了状态数据共享,又提高访问效率。目前比较流行的缓存系统有 Memcached 和 Redis,下面将分别介绍。

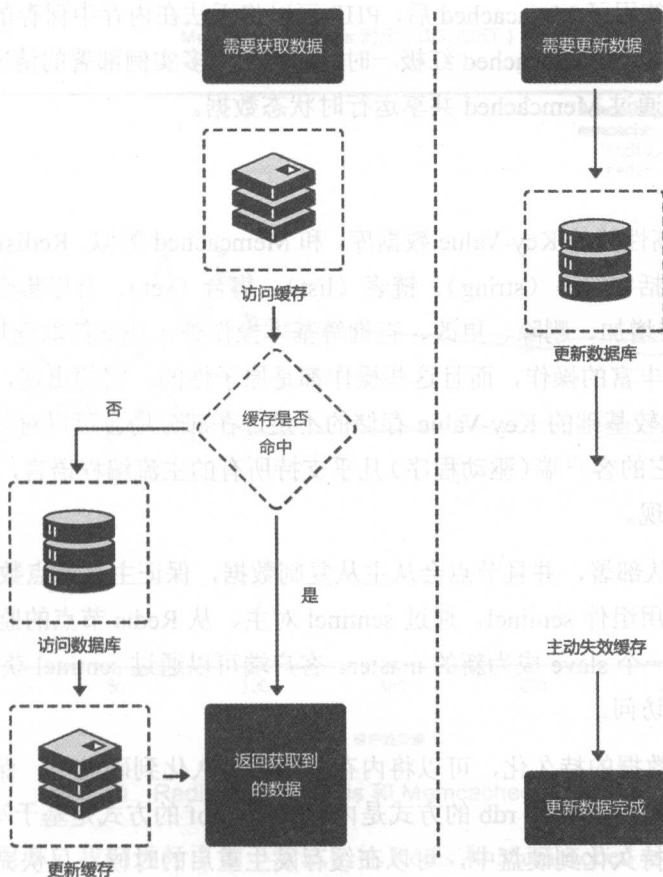


图 3-9 带缓存系统的数据库访问流程

Memcached 简介

Memcached 是一个高性能的分布式内存对象缓存系统,用于动态 Web 应用以减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数,从而提高动态、数据库驱动网站的速度。Memcached 服务器端内部可以理解为一张巨大的 Hashmap,数据保存在内存中,通过网络访问。Memcached 守护进程(服务器端)是用 C 语言实现的,而客户端(驱动程序)几乎可以支持任何语言,并通过 Memcached 协议通信。

Memcached 可以用于数据库的缓存系统，把数据库热点数据（如上面说的商品基本信息）保存在 Memcached 中，大大减少系统读取数据的时间。Memcached 一出现就和 PHP 结合得非常紧密，使用了 Memcached 后，PHP 可以将无法在内存中保存的临时数据保存到 Memcached 中，这也让 Memcached 红极一时。同样，在多实例部署的情况下，多个应用服务部署的实例可以通过 Memcached 共享运行时状态数据。

Redis 简介

Redis 是一个高性能的 Key-Value 数据库，和 Memcached 类似，Redis 支持存储的 value 类型更加丰富，包括字符串（string）、链表（list）、集合（set）、有序集合（zset）、哈希表（hash）等，除支持增加、删除、更改、查询等基本操作外，还支持取交集、差集和并集，以及排序等其他更丰富的操作，而且这些操作都是原子性的。它的出现，很大程度补偿了 Memcached 这类比较基础的 Key-Value 存储的不足，在部分场合可以对关系型数据库起到很好的补充作用。它的客户端（驱动程序）几乎支持所有的主流编程语言，其通信协议 Redis 也很简单，易于实现。

Redis 支持主从部署，并且节点会从主从复制数据，保证主从节点数据的一致。Redis 官方还自带了高可用组件 sentinel，通过 sentinel 对主、从 Redis 节点的监控，在 master 出问题后，自动选取一个 slave 成为新的 master。客户端可以通过 sentinel 获取最新的 master，从而提供高可用的访问。

Redis 还支持数据的持久化，可以将内存中数据持久化到硬盘中，保证数据不丢失。Redis 提供了两种持久化机制，rdb 的方式是内存快照，aof 的方式是基于写操作日志的。通过将缓存中的数据持久化到硬盘中，可以在缓存发生重启的时候，尽快完成缓存预热，而不会因为缓存重启造成所有缓存的数据全部失效。

Redis 发展到 3.0，还提供了集群功能，多个 Redis 节点可以组成一个集群，同时对外提供服务，每个提供服务的节点还可以有自己的 slave 节点，在节点宕机后，slave 会顶替它的位置，从而提供可伸缩、高可用的集群服务。

Memcached 和 Redis 比较

下面分别从性能、内存利用率、监控信息、功能对比和扩展性 5 个方面分别比较 Memcached 和 Redis。

- 性能: 图 3-10 是 Redis 作者对 Redis 和 Memcached 性能比较结果, 可以看到两者性能不分伯仲。

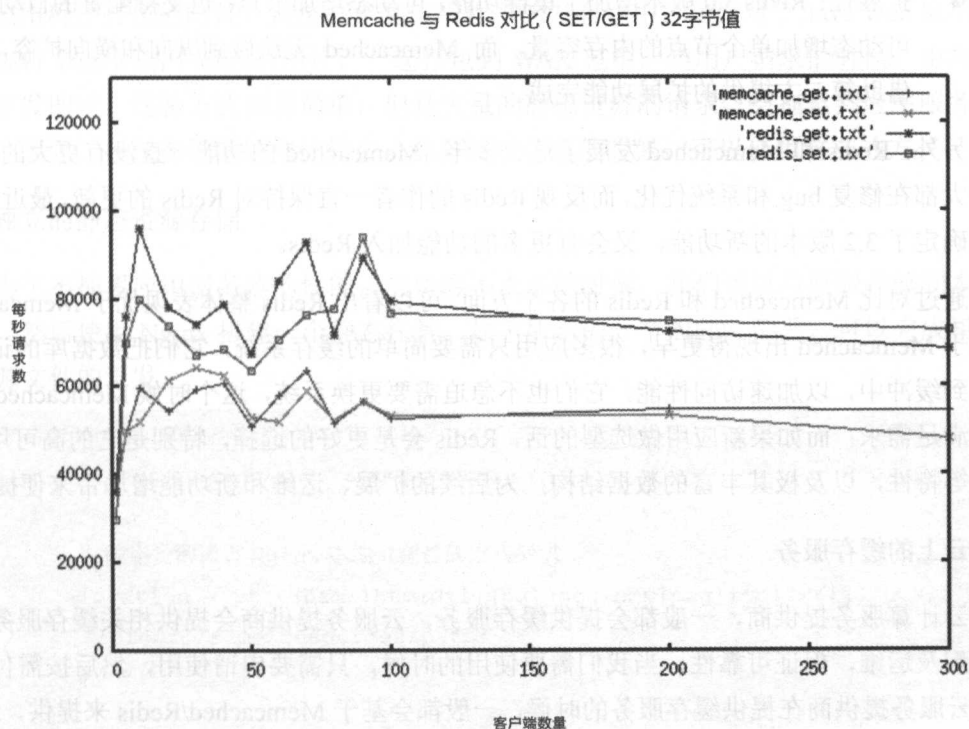


图 3-10 Redis 作者对 Redis 和 Memcached 性能比较

- 内存利用率: 我们进行了如下测试, 往 Redis 和 Memcached 分别插入 50000 条记录, key 为 “test_” 加上 0 到 50000 的编号, value 为 1024 个 x, 即 1KB 数据。实际 Redis 使用的内存为 68M, Memcached 使用内存为 58M, 由于系统本身会使用一些内存和每条记录还有些其他信息, 所以实际使用的内存会大于 key 加上 value 使用的内存总数。Redis 由于数据的信息比 Memcached 多, 因此使用了更多的内存, 但总体来说, 内存利用率表现都非常不错。
- 监控信息: Redis 的 info 命令和 Memcached 的 stats 命令都能监控系统的运行状态, 两者都提供了非常丰富的监控数据。
- 功能对比: 前面在介绍 Memcached 和 Redis 命令的时候就能看出来, Redis 提供

了比 Memcached 更多的功能，除了具备 Memcached 的 `string` 字符串功能外，还支持更多的数据结构、更丰富的数据操作，有很多类似数据库的功能。

- 扩展性：Redis 3.0 版本增加了集群功能，可动态增加节点；也支持配置的自动修改，可动态增加单个节点的内存容量，而 Memcached 无法做到纵向和横向扩容，只能借助第三方提供的扩展功能完成。

另外，Redis 和 Memcached 发展了这么多年，Memcached 的功能一直没有更大的突破，更新大都在修复 bug 和系统优化，而反观 Redis 的作者一直保持对 Redis 的更新，最近 Redis 已经确定了 3.2 版本的新功能，又会有更多的功能加入 Redis。

通过对比 Memcached 和 Redis 的各个方面，可以看出 Redis 整体表现优于 Memcached，但由于 Memcached 出现得更早，很多应用只需要简单的缓存系统，它们把数据库的记录序列化到缓冲中，以加速访问性能，它们也不急需需要更换系统，这个时候 Memcached 基本上能满足需求；而如果新应用做选型的话，Redis 会是更好的选择，特别是它的高可用、高可靠等特性，以及极其丰富的数据结构，为后续的扩展、运维和新功能增加带来便捷。

云上的缓存服务

云计算服务提供商，一般都会提供缓存服务，云服务提供商会提供相关缓存服务的资源分配及运维，保证可靠性，当我们需要使用的时候，只需要申请使用，然后按需付费即可。云服务提供商在提供缓存服务的时候，一般都会基于 Memcached/Redis 来提供，这样，用户在使用的时候对应用程序几乎不用改动，包括 AWS、阿里云及网易云在内都提供了类似的缓存服务。网易云提供了兼容 Redis 协议的线上 Key-Value 缓存服务，支持 Redis 的主从热备，自动容灾，当主节点发生故障的时候，可以实现秒级自动切换，同时也会提供高质量的运维服务保证。

3.1.4 静态资源存储

图片、视频、文档等类似的数据是互联网应用非常基本的组成元素，这些数据都是典型的非结构化数据，不便在关系型数据库中存放。由于这些数据往往都是在运行时不变的，不是由应用程序生成的，也不需要被修改，可以无差别地投递给所有的用户终端，因此又称之为静态资源。本节将如何介绍对这类数据进行存储及管理。

与应用程序一起打包

如果静态资源的量不大，比如可能就是简单页面中的几张图片，在这种场景下，一般说来应用程序打包的时候，直接打包到应用程序的发布包中。比如对于 Java Web 应用，可以直接将 Web 网站中用到的图片，一起打包到 WAR 包中。当用户请求的时候，由应用服务器下发即可。这种方式部署简单，但是大量的静态资源的请求，会拖累到应用服务器的性能。因此，仅仅在资源访问比较少，业务比较简单的时候，可以使用这样的方式。

独立的静态资源存储

为了不拖累应用服务器的性能，当资源不太多的时候，我们可以将资源存储到本地服务器，然后使用 Nginx 搭建一个静态托管。如下所示的一个 Nginx 配置，可以完成相应静态资源文件的下发。

```
server {  
    listen      80;      server_name  ~;  
  
    # 静态资源通过 Nginx 本地资源盘的方式提供  
    location ~ .*\. (html|htm|gif|jpg|jpeg|bmp|png|ico|txt|js|css)$ {  
        root /some/resources/path;  
    }  
}
```

如果此类静态资源非常多，由于本地服务器无论是存储容量还是存储可靠性方面的限制，传统的 IT 公司往往会使用 NAS（Network Attached Storage）或者 SAN（Storage Area Network）这样的传统存储方案来解决大规模的图片、视频存储问题。在规模较小的情况下，购买普通 NAS 设备成本往往相对较低，大概需要几千元人民币；但是规模大到上百 TB 或者 PB 级别就需要采购几十万甚至百万级别的集群 NAS 或者 SAN。

云环境下的静态资源存储

云计算存储服务的推出，特别是对象存储服务的推出，大大方便了图片、视频等静态资源数据的存储、处理和访问。

国内外绝大部分云计算服务厂商都提供对象存储服务，比如 AWS S3、网易 NOS、阿里 OSS 等，国内厂商除了提供了海量、高可用、高可靠的云存储基础服务外，还会提供丰

富的图片、多媒体在线处理服务，所有服务由平台无关的 HTTP RESTful API 接口提供。一站式解决互联网时代图片、视频、文档等非结构化数据管理难题，助力产品实现最佳用户体验。使用云服务提供商提供的对象存储服务相比传统的 NAS 或者 SAN 方案有无可比拟的优势。

- 按需服务：对象存储服务平台提供海量的存储空间和横向的扩张能力，时刻满足用户不断增长的存储空间和访问需求，再也不用一次性购买昂贵的存储设备，不用考虑下次扩容该购买多少设备等烦琐的事情。
- 便捷服务：传统的 NAS 设备一般通过本地文件系统 POSIX API 的方式提供访问，要构架对外的存储和数据访问能力，用户还要搭建一整套上传下载的程序才能够对外提供服务。对象存储系统提供全方位的 SDK，方便开发者使用各种语言进行快速接入。并且使用 HTTP RESTful API 接口对外提供访问操作存储资源的能力，使得外部用户只要有网络，就可以直接通过对象存储服务访问图片、视频等资源。直接把大流量的访问旁路到对象存储服务，再也不用担心对外的带宽不够用或者被大流量的 DDoS 攻击。
- 安全稳定：数据多重备份，用户文件多重备份保障，任何一台服务器或硬盘有故障时，将立即进行数据恢复，确保数据安全毫无隐患。通过完善的权限控制体系、身份验证机制确保数据安全，杜绝未授权访问。支持 HTTPS 方式上传和下载文件，确保数据传输安全可靠。
- 增值服务：针对图片、视频等资源，我们往往需要对外提供丰富的处理服务。构建这些服务需要使用很多的图片处理音视频处理软件，比如 ImageMagick、FFmpeg 等，玩转这些开源组件并对外提供稳定可靠的服务体系不是一件容易的事情。网易对象存储服务 NOS 提供了非常完善的图片处理，支持图片的缩略、裁剪、图文水印、旋转、质量参数、模糊化、类型转换、渐进显示等功能。同时也支持丰富的音视频处理，支持音视频的元数据获取、音频转码、视频定位拖拽、视频帧截图、截图去黑边、黑帧智能判断等功能。

总而言之，非结构化的静态资源存储服务的选型与具体的业务类型关系紧密。如果是对象存储密集型的，比如视频或者文档分享这样的应用，或者电商这种需要存储大量商品图片的应用，推荐使用专业的云对象存储服务。对于静态资源不会很多，而且相对固定的应用，可以考虑自己通过 Nginx 搭建一个静态资源托管服务，甚至直接与应用程序一起打包。

3.2 架构实践

在确定好技术选型后，就进入具体的实践阶段。本节主要介绍在实践阶段如何快速迭代开发、快速交付部署、实现初步的高可用。同时我们的实践从一开始就要注意安全风险，避免写出有安全漏洞的代码。

3.2.1 快速迭代

在初创期，一般业务不是很稳定，往往会不停地增加新功能。因此，这也要求我们的应用能够满足快速迭代的需求。

为了快速地进行功能的迭代，除了在需求管理和团队沟通优化上做足文章外，对于业务功能的具体实践者，也就是我们的开发人员来说，无非就是在接到需求，理解需求的情况下，能够快速完成功能的开发并且上线。

快速搭建开发环境

当有新人加入团队及团队有人更换开发环境的时候，传统的搭建开发环境的方式，一般要安装不同的软件，如数据库、语言虚拟机（如 JVM 之于 Java）和一些依赖的三方库等。结果经常出现开发环境搭建出错，无法正常运行，浪费了我们的时间。

究其原因主要是我们的项目，除了我们在依赖配置文件中指定的一些三方依赖及我们应用程序的本身配置以外，实际上还隐式地依赖了运行环境及运行环境的相关配置，不同的运行环境，由于版本的不同，底层系统的不同，从而导致我们无法快速地完成一个开发环境的搭建。

云原生应用的 12 要素里面就有一条，要求显式地声明依赖关系，这里的依赖不仅仅是应用的依赖，还包括对系统级的依赖项。使用基于 Docker 镜像的方式，可以快速开发环境搭建。在 Docker 镜像中，我们可以将一个应用运行时的所有依赖，都打包到镜像中，这些依赖不仅包括程序使用的三方库，还包括系统级别的依赖，比如依赖的 libc 库版本，使用的一些系统级别的工具等。这样，当搭建开发环境的时候，我们只需要将镜像拉下来，就可以获得一个包含了所有依赖项的开发执行环境，在这个镜像基础上开发，保持了开发环境的一致性，同时也节省了搭建环境的时间开销。

代码模块化

很多时候，为了保持新功能快速迭代，我们可能会做如下的事情。

- 当我们需要加新功能的时候，往往是直接在原来的代码库中，找到一个可以加代码的地点，然后将代码加上，而不去考虑代码加的位置是否合适。
- 如果新增加的代码与原来已有的某个代码片段功能类似，直接将原有的代码片段复制过来，然后进行少量修改，甚至不用修改仅仅是复制代码，造成内部代码的大量重复。

上述行为，短期内看是合理的，它可以带来新功能的快速迭代，然而从长期来看，伤害了一个项目快速迭代的可持续性。

在代码中不合适的地方加入新的代码，可能导致业务代码耦合严重，不同的业务逻辑的代码全部糅合到一块，当后续再有新的功能需求的时候，导致另外一处与新功能本来不相干的功能出现问题，就好比给汽车换了个发动机，结果轮胎出问题了。

大量的代码直接进行复制，同样也会带来问题，假如被复制的代码出现了 bug。在进行修复的时候，我们需要在所有复制的地方都进行修复，一旦有某些地方出现遗漏，会导致 bug 依然存在，代码的可维护性变得很差，严重拖慢我们的迭代速度。

模块化做好的代码，模块之间低耦合，只要保持模块对外的接口稳定，其内部实现可以独立进行更新迁移。阅读代码的时候，也更易理解，更容易维护，不会导致修改了一个模块，在另一个不相干的地方出现问题的情况。同时相对独立的模块，往往可以在多个地方被用到，提高了代码的可重用性，也更易于测试。当发现某一个模块的性能出现问题后，我们还可以直接将这一个模块抽出并进行独立部署，为后续的服务化转变打下基础。

为了做好模块化，我们可以先对业务进行分层，最典型的是使用 MVC 模式的时候，模型和控制器之间有明显的分界线，它们可以仅仅通过接口进行耦合，面向接口编程。保持接口的稳定，即使我们的底层数据库选型改变了，只要模型层的接口不变，控制器的代码就不用改变，需要修改的仅仅是模型层代码具体实现的配置。

在模型层的代码中，两个完全不同的业务之间也可以划分到不同的模块中，分别进行开发及测试。同样，在控制器的代码中，毫不相干的业务逻辑之间也可以划分到不同模块中。

使用前面所说的控制反转及依赖注入框架，在具体编程的时候，仅仅面向接口编程，不在代码中硬编码具体的实现类。通过配置文件的方式，将具体对象的实例化及依赖注入延迟到运行时，提高代码的可扩展性。

对于一些业务中可能出现相似的代码，不要进行代码的复制，要进行抽象，可以考虑通过增加一个参数，或者增加配置项的方式来把这些相似的代码做统一，避免代码中的硬编码。始终要牢记的一点就是我们的代码中，尽量多提供机制，而具体的策略选择，可以考虑作为配置项的形式来提供。最常见的例子，就是很多时候程序中需要某一个参数，这个参数往往是一个经验值，此时将参数作为配置项比直接硬编码到代码中修改简单。

有时候，由于业务的需要，或者由于业务上线时间紧迫，可能还会临时做出一些不符合模块化的行为，比如需要临时加一个功能，这个功能又与已有的某一个功能相似，直接复制已有的代码，然后再做简单修改会比较快，我们往往也会这样做来达到快速的功能上线。这样做短期内是没问题的，但是在线上以后，后续的版本中一定要将这段代码进行重构，消除重复代码。

根据破窗效应，一旦项目中出现了不良代码，又没有及时重构，后续可能就有更多的不良代码出现，就好比一个有少许破窗的建筑，如果破窗没有及时被修复，将会有更多的破坏者破坏更多的窗户。

完善的测试

一个比较好的模块化系统，其模块内部高内聚，模块间松耦合。我们要在模块内部做好单元测试，模块之间对接口做好测试。测试用例尽可能完善，能覆盖到尽可能多的情况。每当我们完成新功能开发的时候，都要针对新功能加测试代码，时时保证测试用例集比较高的代码覆盖率。只有测试完善了，无论是新功能增加，还是 bug 修复，我们都可以快速回归测试，以确定是否真正达到需求预期，从而提高我们开发迭代的速率。

上面讲到了如何在搭建环境阶段和开发测试阶段保持快速迭代，除了这些，还必须要有比较快速的交付及部署方式作为保证。在 3.2.3 节，我们会着重介绍如何在云原生架构下，快速完成交付及部署。

3.2.2 高可用与负载均衡

当服务上线运行后，要尽量减少服务中断时间，也就是说让系统有较高的可用性。高

可用往往与集群一起出现，因为高可用是由资源冗余来实现的。高可用本身又是一个很大的话题，从硬件资源的高可用，到依赖系统的高可用，再到具体应用设计的高可用。在产品的初创阶段，高可用指标也不用太苛刻，我们只需要实现最基本的高可用，即用多点部署来达到基本的可用性保障。

高可用概念

衡量一个系统的可用性，使用的指标是给定时间内可用时间占用的百分比。这里可用的意思是指用户可以正常地使用系统，比如，即使系统没有瘫痪，但是对用户来说响应延迟远远超出预期，那么我们也可以认为此时系统处于不可用状态。系统的可用性反映了系统的稳定程度。

高可用系统是指一个系统的可用性比较高，也就是说可用时间占用总体运行时间的百分比比较高。一般我们会使用“N个9”这样的术语来描述高可用指标，“1个9”就是90%，“2个9”就是99%，以此类推。为了达到某一个高可用需求，就要尽可能降低系统的宕机时间。表3-1列出了不同的可用性百分比下，一年的最大宕机时间。

表 3-1 不同可用性下的最大宕机时间

| 可用性% | 一年宕机时间 | 一月宕机时间 | 一周宕机时间 | 一天宕机时间 |
|-----------------|----------|---------|---------|---------|
| 90% (“1个9”) | 36.50d | 72h | 16.8h | 2.4h |
| 99% (“2个9”) | 3.65d | 7.2h | 1.68h | 14.4min |
| 99.9% (“3个9”) | 8.76h | 43.8min | 10.1min | 1.44min |
| 99.99% (“4个9”) | 52.56min | 4.38min | 1.01min | 8.66s |
| 99.999% (“5个9”) | 5.26min | 25.9s | 6.05s | 864.3ms |

高可用的意义

下面，我们以一个10万PV的产品为例，来看看使用简单的异常→通知→人工处理异常模式会造成的影响。

- 异常发现阶段：服务发生了异常，我们假定30s内触发了相应的监控报警，经过系统处理，30s之后（考虑到程序处理和运营商的延时，30s已经是一个比较理想的数字）发送到相关负责人的手机上。我们假定这个负责人十分负责，并且会及时查看手机信息，因此他迅速查看了手机，发现了系统出现问题。假定他看信息

所花时间为 10s, 那么从异常出现到运维人员得到通知, 至少得 1min 的时间。

- 故障定位阶段: 我们再假定报警短信十分详备, 运维人员经验也非常丰富, 并且快速地完成了故障定位, 花费时间假定为 1min。
- 问题处理: 假设对应负责人手边恰好就是他的工作电脑, 有现成的网络环境, 迅速连接到出问题的服务器上, 并且找到对应的进程, 完成进程重启或者配置调整。这个过程假定时间 1min。

我们完成了所有的故障处理后, 服务开始恢复。假定服务通过简单的重启就可以恢复, 时间为 30s。

最终, 我们的服务恢复了, 在最好的情况下用了差不多 5min 的时间。按照产品高峰期 10h 占 90% 的流量, 其他时间占用 10% 的流量来算, 在业务高峰期, 平均每秒的访问人数为 $90000/10/3600 = 2.5$ 人, 那么 5min 的服务异常时间, 差不多会有 600 位用户受到影响。

但是上述都是假设在理想情况下, 实际上任何一个环节都可能出现意外, 运营商的报警信息延迟、运维人员没有实时看报警信息、运维人员手边没有网络、运维人员没能快速定位到错误所在等。从互联网产品运维的经验出发, 在相关人员都是正常水平从业专业人员的情况下, 15min 已经是非常理想的情况, 按照这个标准, 高峰期影响的人数可能达到 2000 人左右。而且一般触发异常的情况, 都是业务高峰期, 只有在业务高峰期, 才会对系统带来压力, 出现一些平时测试时没有遇到的情况; 反而业务低谷的时候, 更不容易出现异常。

一个产品的可用率, 不但会直接影响到用户业务, 还会直接影响用户对产品的信心。如果说服务不可用对用户造成的经济损失随时可以通过代金券、免费物品等经济方式来弥补, 那服务不可用造成对产品信心的损失则几乎不可恢复。特别是在初创期, 获取用户已经非常不容易, 如果由于服务的接二连三不可用导致用户流失, 更加得不偿失。

因此, 我们不仅要解决服务能不能恢复的问题, 还需要解决服务如何快速恢复, 并且在服务节点部分失效的情况下保证系统仍然可以正常工作的问题。

一个高可用的系统会为我们带来最长恢复时间的保证, 由于没有人为介入, 异常恢复处理是自动进行的, 因此保证系统的最长恢复时间, 可以大大减少对用户的影响。

另外, 由于一般的请求都有重试机制, 因此在使用高可用的情况下, 当单个节点异常时, 对用户来说, 往往是出现短暂的延时, 而不会失败。即使没有重试机制, 一般用户也会通过重试来尝试解决问题。由于服务可以快速恢复, 在用户重试几次之后就会发现服务

恢复正常，在这种情况下，用户并不会认为是服务的问题，觉得只是单纯的网络抖动或者其他原因，并不影响用户对产品的信心。

系统健康检查

先要知道系统是不是健康可用的，才可能在出问题的时候进行自动恢复。传统上，如果一个系统的进程无法工作，或者端口监听不在了，就肯定不健康，也很容易被检测到。

实际上，更多情况下，我们要面对的不健康情况，是由于系统的一些异常情况处理不严谨或者随着负载增加，造成系统的响应不满足预期。比如，由于系统的 bug，使得系统运行进入死循环，此时虽然进程没停止，端口监听完好，但是却无法对外提供服务。再比如，某一个页面，正常情况下加载会在几秒内完成，业务要求在 10s 内完成。如果由于系统问题，页面加载需要 1min，那么此时虽然页面还能加载完成，但是达不到业务需求，系统也是不健康的。

因此，这里的健康检查，实际上更多是对系统的业务健康度的健康检查，往往与业务有关。一般情况下，这种场景下的健康检查有两种方式：脚本和协议。

脚本的方式，是每隔一段时间运行一次某一个定制的本地脚本，根据脚本的返回码确认系统是否健康。在脚本执行中，可以去检查进程状态，发一些模拟请求等。还有一种比较常见的方式，就是在系统中插入一些检查点，然后在脚本中对这些检查点进行检查。比如，可以在系统中加入，每隔一段时间，就把当前时戳写到一个本地文件中，然后在脚本中对这个文件进行检查，一旦发现这个本地文件更新不正常，就可以认为原来的系统已经不健康了。

对于绝大部分应用来说，一般都是 Web 服务，Web 服务直接基于 HTTP 协议进行健康检查更合适。比如，对于一个论坛，每隔一段时间，请求一次其主页，看看是否能够请求成功，响应时间是否满足其需求。对于 HTTP 健康检查，需要配置其健康检查的 URL 及响应超时时间。对于一个长连接应用来说，配置 TCP 的健康检查，需要指定端口号及连接建立超时时间。

高可用实现方式

在初创期，我们的服务架构是一个单体架构，这里的高可用主要是指我们服务器进程的高可用，一般是采用服务器冗余来实现的，当部分节点异常时，其他节点仍然可以正常工作，从而保证系统正常运行。同时，多点部署也能够实现初步的水平扩展，以应对用户

的增长。

多个节点的服务部署构成一个集群，也就是高可用集群，这也是我们在阅读一些文章的时候，会发现高可用总是与集群一起出现的原因。

在一个高可用集群中，我们可以配置对各个节点的健康检查，当有节点出现异常的时候，只要其他节点还在正常工作，系统整体的可用性就不会受到影响。我们可以将系统部署在容器中，通过 Kubernetes 进行管理，多点部署实际对应的就是 Kubernetes 中的多个副本的概念。Kubernetes 同时也支持对其上运行的容器进行健康检查，并且可以在健康检查失败后，自动重启相关的 Pod。

除了系统 bug 过多导致系统频繁异常，造成自我 DoS（Denial of Service，拒绝服务）攻击外，使用这种多点部署的方式，基本上可以保证“2个9”到“3个9”的可用性。

负载均衡

在对系统进行多点部署后，我们需要将用户请求按照不同的转发策略，发送到部署的不同的节点上，负责完成请求转发的设备，我们称之为负载均衡。负载均衡可以完成流量分发，负载均衡的服务示例如图 3-11 所示。

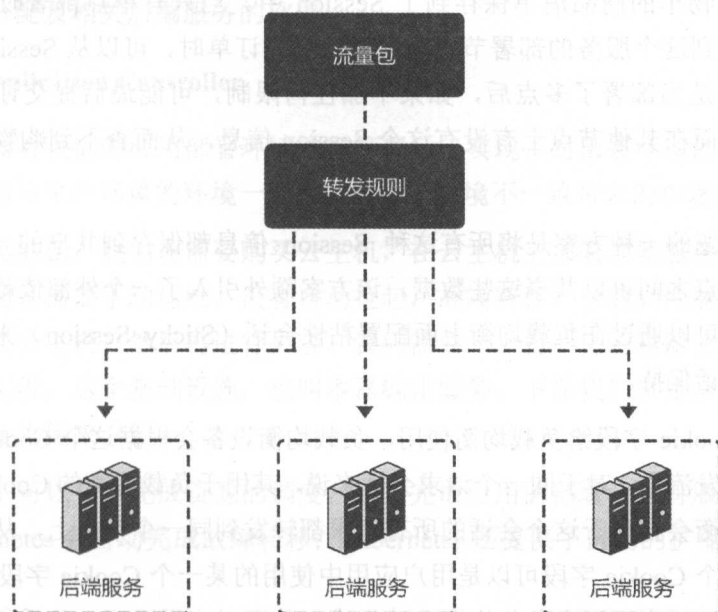


图 3-11 负载均衡服务示例

负载均衡作为用户服务的统一入口，将用户流量分摊到不同的后端服务上去，可以实现服务的初步水平扩展，以应对支持增长的业务压力。另外，负载均衡设备除了提供流量分发外，还会提供故障隔离。一般负载均衡都提供了对后端服务的健康检查机制，如果检测到后端节点异常，负载均衡可以不再向其转发请求，完成故障隔离，提高系统的可用性。

对于 Web 应用来说，我们一般可以使用 Apache、Nginx 等来实现负载均衡。云原生架构下，可以直接使用这些软件的镜像，一键部署，即可启动一个负载均衡节点。不过，假如我们使用 Nginx 来实现后端节点的负载均衡，那么我们的 Nginx 又成为了单点，我们不得不再启用多个 Nginx 节点来进行高可用，就算我们完成了 Nginx 的多点部署，交换机也可能挂掉。因此，自己去实现一个高可用的负载均衡是困难的。不过，幸运的是，在云计算中，基本上所有的云服务提供商都把负载均衡作为基础服务提供，如 AWS 的 ELB、阿里云的 SLB、网易云基础服务的 NLB 等。由于负载均衡服务偏底层，因此不同云计算服务商提供的负载均衡服务虽然底层实现技术各不相同，但是实现的功能都是类似的。

粘性会话（会话保持）

如果一个服务是有状态的，服务中维护了用户某一系列请求的 Session 信息，以电商为例，假如把购物车的商品清单保存到了 Session 中。当只有单点部署时，用户的每一次请求，都会落到这个服务的部署节点上，最后提交订单时，可以从 Session 中直接获取购物车信息。但是当部署了多点后，如果不加任何限制，可能最后提交订单的请求落到了其他节点上，而在其他节点上有没有这个 Session 信息，从而查不到购物车信息，导致问题。

解决这个问题的一种方案是将所有这种 Session 信息都保存到共享的一个外部缓存服务中，这样多节点之间可以共享这些数据。该方案额外引入了一个外部依赖，从而引入了复杂性，我们还可以通过在负载均衡上面配置粘性会话（Sticky Session）来实现，粘性会话有时也称作会话保持。

使用一个 Cookie 字段给负载均衡使用，负载均衡设备会根据这个 Cookie 字段，来决定往哪个节点转发流量。对于同一个请求会话来说，其用于负载均衡的 Cookie 字段值都是相同的，负载均衡会把属于这个会话的所有请求都转发到同一个节点上，从而避免了上述提到的问题。这个 Cookie 字段可以是用户应用中使用的某一个 Cookie 字段，比如 Tomcat 中的 JSESSIONID，也可以由负载均衡自动生成，从而对应用透明。几乎所有的云负载均衡

衡服务都提供了对粘性会话的支持，我们只需要对负载均衡进行一下配置即可。

3.2.3 交付与部署

传统的软件部署方式中，应用软件包开发完成后交付给运维人员，运维人员需要准备服务器资源，部署应用程序，部署相关的依赖如数据库、缓存等，这种部署方式极其烦琐，不够灵活。在基于云原生架构中，交付和部署变得非常简单。

后端服务

云计算服务商提供常见的后端依赖，如关系型数据库、缓存服务、对象存储、负载均衡等服务，都是开箱即用的。直接使用这些云服务，避免了我们自己搭建这些后端依赖服务，降低了部署的复杂性。这些服务往往都是按需计费，有专业的运维团队保证数据的可靠，保证服务的高可用，比起我们自己手动搭建的后端服务，无论是稳定性，还是性价比方面，直接使用云计算服务商提供的云服务都是比较不错的选择。

所以，特别在初创期应用部署的时候，我们往往不需要考虑依赖的后端服务的部署，只需要考虑好自己的应用部署，然后直接购买相应的后端资源即可，由云计算服务商来完成相应资源的分配及相关后端服务的部署。

Pods 与 Replication Controller

在使用容器对我们的应用部署环境进行管理后，实现了对部署环境的版本化控制，也保障了测试环境与生产环境的环境一致性，消除了环境不一致带来的部署时的问题。

仅使用容器的话，我们还需要购买云主机，在云主机上部署容器服务。一旦云主机出现问题，我们还是需要手动运维，没有自动拉起，故障转移。当遇到性能瓶颈的时候，还需要手动去购买云主机，然后再进行扩容。我们需要一个自动化的容器管理基础设施，来改善对容器的运维，这个基础设施，也叫容器编排服务。下面我们目前流行的容器编排服务 Kubernetes 为例来介绍。

Kubernetes 可以自动完成资源的调度，快速完成应用的部署；当有服务节点发生故障的时候，Kubernetes 会自动完成故障转移；Kubernetes 还提供了自动的扩缩容、无缝滚动升级等特性。

Pod 是 Kubernetes 中的一个概念，一个 Pod 可以是一组紧密耦合的容器（比如 Docker

容器)的集合,也可以仅包含一个容器,这些容器都运行在同一个物理机或者虚拟机上。一个 Pod 中的所有容器都运行在一个共享的上下文中,共享网络命名空间,持久化地卷存储等资源。Pod 与 Pod 之间资源互相隔离,大体上,我们可以认为一个 Pod 是一个“逻辑主机”。

Pod 是 Kubernetes 进行资源调度及资源分配的最小单元,通过提供了 Pod 这一层的抽象,而不是直接去管理容器,带来了灵活性,也可以去适应更多的场景,比如传统的 LAMP,我们就可以考虑把相关的 Apache、MySQL 等作为不同的镜像运行在一个 Pod 中。Pod 不会持久,我们可以认为其仅仅在运行时存在,一旦被删除了,Pod 中的所有数据就都不在了,如果有数据持久需求,需要在启动相关容器的时候,挂载外部的文件系统,以达到将产生的数据持久化到外部存储。

实际上,我们往往不会去直接操作 Pod,而是通过 Replication Controller 来对 Pod 进行管理,Replication Controller 提供了集群范围内的 Pod 复制及调度的能力,它保证了在所有时间内,都有特定数量的 Pod 副本在运行,如果 Pod 副本太多了,就杀死几个,如果太少了,就多创建几个。和直接操作 Pod 不同,Replication Controller 监管着整个集群中所有节点上面运行的 Pod,自动完成替换掉那些被删除或者被终止的 Pod。因此,哪怕我们仅仅运行了一个 Pod 的副本,最好还是使用 Replication Controller 来管理,因为我们的 Pod 可能会因为出现异常而终止,而使用 Replication Controller 管理后,一旦 Pod 出现异常,Replication Controller 就会创建出新的 Pod 运行实例来替换出现问题的 Pod。

应用部署

我们的应用在部署的时候,可以使用基于 Pod 和 Replication Controller 的方式来进行。我们仅仅需要配置好 Pod 需要的容器镜像及版本,设定好要运行几个副本,然后将其提交给编排服务即可。

使用这样的部署方式,极大地提高了部署效率。当我们需要高可用部署和多点部署的时候,只要指定部署几个副本,编排服务自动帮我们完成资源分配调度并按照我们的需求部署运行多个副本。当有部署的节点由于硬件故障或者网络原因出现故障时,编排服务又会自动将出问题的节点删除并创建新的节点进行替换。

滚动升级及回滚

当需要对应用进行升级的时候，容器编排服务提供了自动的滚动升级机制。我们只需要更新一下相关配置中的 Docker 镜像版本，即可触发编排服务的自动滚动升级。滚动升级的时候，Replication Controller 会先创建出一个使用新的镜像版本的 Pod，然后删除一个使用旧版本镜像的 Pod，并一直重复这个过程，直到旧版本的 Pod 全部被删除为止，完成升级更新。

如果在升级过程中或者在升级结束后，我们发现了新版本的问题，就可以直接回退 Pod 使用的容器镜像版本，Replication Controller 在获知这一改变后，会像滚动升级过程那样，对升级过程进行回退，直到所有的 Pod 全部回退到旧版本的容器镜像为止，完成版本的回滚。

网易云基础服务

网易云基础服务基于 Kubernetes 提供了开箱即用的容器编排服务，作为企业级的容器云平台，支持针对应用集群的一键部署，云计算资源可以弹性扩展。上述所介绍的关于容器部署过程及应用的滚动升级、回滚操作，都可以很方便地在网易云基础服务中进行，这提高了交付及部署效率，使得我们的功能开发在初创期可以快速地迭代开发上线。

3.2.4 Web 应用安全

目前很多应用的业务，都提供一个 Web 服务，Web 服务很常见，因此各类针对 Web 的漏洞发掘和渗透攻击也越来越多。黑客利用网站操作系统、中间件、Web 代码的漏洞进行攻击，从而获得 Web 服务器和数据库服务器的控制访问权限，进行页面篡改、挂马、暗链、数据泄露、肉鸡等一系列非法操作。

在业务部署到云上面之后，Web 安全问题并未得到缓解。作为一个技术人员或者架构人员，了解必要的 Web 安全技术掌握其原理和防护方法，显得尤为重要。接下来我们简单介绍一些 Web 安全的各种问题和对策。

1. XSS 漏洞

XSS 是 Cross Site Scripting 的简写，即跨站脚本攻击，为了避免和层叠样式表(Cascading Style Sheets, CSS)的缩写混淆，故缩写为 XSS。

恶意攻击者往 Web 页面里插入恶意 HTML 代码，当用户浏览该页时，嵌入 Web 里面的 HTML 代码会被执行，从而达到恶意攻击用户的特殊目的。按照 XSS 的不同，可以分

为如下几种类型。

反射型 XSS

反射型 XSS 指的是把用户输入的数据“反射”给浏览器，也就是黑客通常需要诱使用户“点击”一个恶意的链接。

攻击者会通过社会工程学手段，发送一个 URL 链接给用户打开，或者在 IM、论坛、博客上发放 URL 诱使用户点击，在用户打开页面的同时，浏览器会执行页面中嵌入的恶意脚本。

存储型 XSS

存储型 XSS 会把用户输入的数据存储在服务器端。当其他用户访问到展示这个数据的页面就会被攻击。

攻击者利用 Web 应用程序提供的录入或修改数据功能，将数据存储到服务器或用户 Cookie 中，当其他用户浏览展示该数据的页面时，浏览器会执行页面中嵌入的恶意脚本。所有浏览者都会受到攻击。

DOM 型 XSS

DOM 型 XSS 的分类并不是按照上面数据是否存储在服务端分类，而是比较特别的一类 XSS，通过修改页面的 DOM 节点形成 XSS。和以上两个跨站攻击的差别是，DOM 跨站是纯页面脚本的输出，只有规范使用 JavaScript，才可以防御。

XSS 漏洞危害

XSS 漏洞可以偷取用户令牌 Cookie，伪造用户身份登录进行恶意操作，这是最常见的。同时，还可以控制用户浏览器，获取用户键盘记录如用户输入的用户名密码，进行扫描内网。比较高级的用法可以结合浏览器自身和浏览器插件的漏洞，下载病毒木马，深入攻击，例如衍生 URL 跳转攻击、蠕虫攻击、DDoS 攻击、盲打后台等。

XSS 漏洞防护

针对 XSS 漏洞的防护，其核心问题在于对用户的输入没有进行验证和过滤，对于可信和不可信边界的划分也是所有安全问题的核心。要针对 XSS 进行防护，需要根据上面不同的攻击类型进行处理。以“谁使用，谁负责”的原则，任何使用用户不可信数据方都需要

对数据进行校验，不信任任何来自客户端的用户输入，对于 XSS 来说，根据不同的应用场景进行过滤。如果是不需要显示富文本的地方，直接对所有的 HTML 转义处理，反射性和存储型可以在后端进行转义后输出到前端；如果是 DOM 型的需要在前端使用 JavaScript 进行转义；如果需要显示富文本，就需要对用户输入进行严格的限制。开源项目 OWASP AntiSamy 是一个可确保用户输入的 HTML/CSS 符合应用规范的 API，更多通过这个开源项目来对 XSS 漏洞进行防护的信息，具体可以参考其官方文档。

2. CSRF 漏洞

Cross-Site Request Forgery (CSRF)，跨站请求伪造攻击，攻击流程如图 3-12 所示。利用这个漏洞可以使受害者在不知情的情况下，向存在该漏洞的 Web 站点发送请求，最终以受害者的身份完成特定操作。CSRF 漏洞的本质原因是重要操作的参数可以被攻击者猜测到。

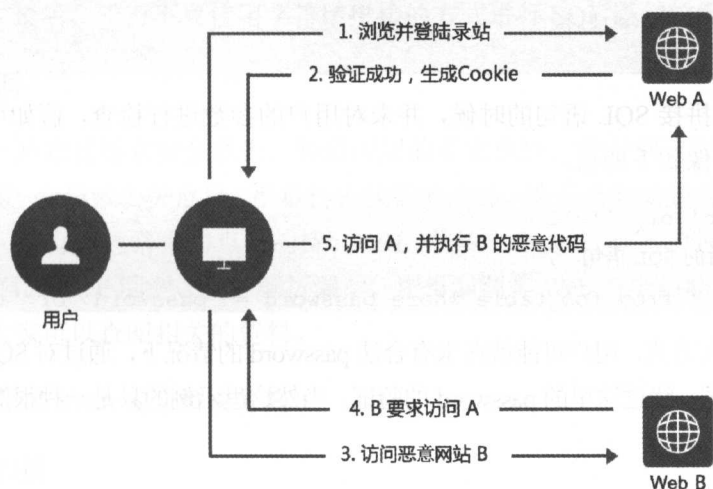


图 3-12 CSRF 攻击示意图

CSRF 漏洞危害

修改用户信息，如用户的头像、发货地址等。更有甚者，可能执行恶意操作，比如修改密码、添加/删除好友或者点赞/转发/评论/私信。

CSRF 漏洞防护

针对 CSRF 漏洞的防护，可以通过在用户提交的表单中加入随机验证值的方式进行防护，还可以使用二次验证，例如使用短信验证码、密码确认等方式进行防护。当然，有些业务为

了不影响用户的体验,可能会使用 `refer` 等字段作为验证,这种方法比较简单但也容易被绕过。

3. SQL 注入漏洞

SQL 注入 (SQL Injection) 是一种常见的 Web 安全漏洞,攻击者利用这个漏洞,可以访问或修改数据,或者利用潜在的数据库漏洞进行攻击。

注入的漏洞本质是代码和数据未分离。通过在用户可控参数中注入 SQL 语法,破坏原有 SQL 结构,达到编写程序时意料之外结果的攻击行为。以下示例存在 SQL 注入漏洞。

```
void doGet(HttpServletRequest req, HttpServletResponse res) {  
    String sql = "select * from foo_table where password = '"  
        + req.getParameter("password") + "'";  
    // 进行查询  
    executeQuery(sql);  
}
```

由于上面在拼接 SQL 语句的时候,并未对用户的参数进行检查,假如用户输入的参数是精心构造的,像如下的值。

```
password' or '1'='1  
// 拼接后的 SQL 语句  
select * from foo_table where password = 'password' or '1'='1'
```

通过这种注入方式,用户可能就在未有合法 `password` 的情况下,通过对 SQL 语句注入,获得不属于它的数据,绕过这里的 `password` 的验证。当然这里示例的只是一种很简单的注入方式。

漏洞分类

漏洞分类分为以下两种。

- 回显注入: 攻击者可以直接在当前界面内容中获取想要的内容或者通过报错信息获取想要的内容。
- 盲注: 数据库查询结果无法从直观页面中获取,攻击者通过使用数据库逻辑或使数据库执行延时等方法获取想要的内容。

SQL 注入漏洞危害

对于 SQL 注入漏洞,最常见的危害就是泄露了数据库中敏感数据给未授权用户,因为

SQL 注入漏洞可以绕过一些认证，可能导致用户名、密码、手机号码、身份证号码等用户信息的泄露。通过一些精心构造的注入手法，可能会获取到管理员的后台密码，甚至获得非法提权和用户系统的权限。

SQL 注入漏洞防护

针对 SQL 注入漏洞的防护，主要是要意识到这种漏洞的存在，不信任任何来自客户端的数据。可以通过过滤一些 SQL 关键字的方式进行防护，比如“select”、“insert”这一类关键字，“`”（反引号）、“””（双引号）、“--”（SQL 注释开头）这一类保留字符等。当然因为各种编码和绕过技术，手工进行过滤是不够的，容易遗漏，比较安全的方式是采用语言预置的功能，例如，用 Java 的 `PreparedStatement` 进行 SQL 操作。当然，现在还有很多框架类的数据库持久化框架，都会自动完成对相关数据的过滤和校验。最后，千万不要使用字符串拼接的方式进行 SQL 语句的组装和操作。

4. 其他漏洞

我们要从一开始就树立安全意识，知道应用的不安全性，并时刻记在心里。一旦发生信息泄露等问题，会导致公关危机，严重伤害我们的产品。除了上面提到 XSS、CSRF、SQL 注入等漏洞外，还有其他常见的点击劫持、URL 跳转、命令注入、文件操作漏洞、XML 注入、XXE 漏洞、SSRF 漏洞、文件解析漏洞、逻辑漏洞等 Web 安全问题。更多的 Web 安全漏洞问题，大家可以查阅相关的资料。

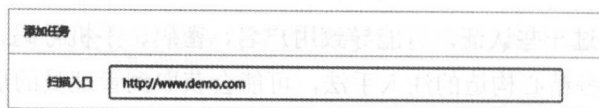
5. 云环境下的 Web 安全解决方案

Web 漏洞扫描

Web 漏洞扫描可以作为检测 Web 漏洞的辅助工具之一，帮我们发现 Web 漏洞。扫描器是通过网络 Fuzzing 功能来探测目标主机和目标服务是否存在漏洞的产品。Web 漏洞扫描可以对 Web 服务器的多种项目进行全面的安全检测，检测 SQL 注入、XSS、CSRF 等漏洞并提供解决方案。

以云扫描器为例，我们使用 Web 漏洞扫描器来对自己的业务进行扫描。一般有以下步骤。

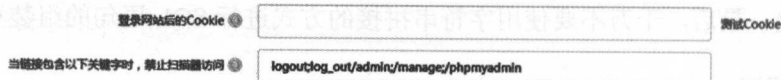
- 在扫描器界面添加需要扫描的域名，例如 `www.demo.com` 作为扫描的入口，如图 3-13 所示。



| | |
|------|--|
| 添加任务 | |
| 扫描入口 | <input type="text" value="http://www.demo.com"/> |

图 3-13 添加扫描入口

- 有一些云平台的扫描器为了防止扫描器被滥用，需要验证任务添加者是否为此域名的拥有者。验证的方式主要分为以下两种。
 1. 文件验证授权，在站根目录下创建文件如“signature.txt”，文件内容为空。
 2. 首页验证授权，在网站首页中嵌入隐藏元素，并带有特定的验证字符串。
- 扫描器以爬虫技术为基础，如果需要深入页面爬取，就需要添加登录信息（一般是有效的 Cookie）和登出页面，如图 3-14 所示。



| | | |
|---------------------|--|----------|
| 登录网站后的Cookie | <input type="text"/> | 测试Cookie |
| 当链接包含以下关键字时，禁止扫描器访问 | <input type="text" value="logout/logout/admin/manage/phpmyadmin"/> | |

图 3-14 添加登录信息

- 为了尽量减少扫描器对 Web 应用的影响，还需要设置扫描的发包速率，防止扫描器对应用产生压力。
- 最后，等待扫描完成，根据扫描结果对响应漏洞进行修复，具体的漏洞原理和修复方式在上面已经进行了叙述，更多内容可以参考 OWASP Top 10 项目，https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project。

Web 应用防火墙

Web 应用防火墙是通过执行一系列针对 HTTP/HTTPS 的安全策略来专门为 Web 应用提供保护的产品，可以帮助用户加固他们的 Web 应用程序，以抵御 SQL 注入、XSS、CSRF、XXE、文件包含漏洞等漏洞。

现在市面上有大量的云 Web 应用防火墙（也叫云 WAF，我们在本书中统一称为云 WAF）可用，而每个云计算平台基本都自带了云 WAF 功能。

- 接入云 WAF 的方式很简单，主要分成两种方式。
 1. 修改 NS 记录的方式，将需要保护的 Web 站点的域名解析 NS 记录改到云 WAF 提供商的高防 DNS 服务，如图 3-15 所示。

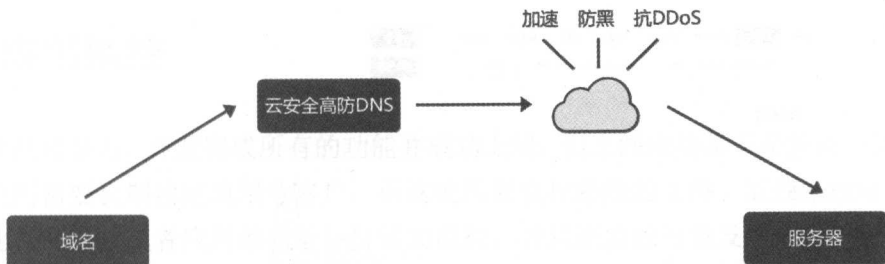


图 3-15 NS 方式接入云 WAF

2. 使用 CNAME 别名的方式，设置将需要保护的 Web 站点的域名删除 A 记录，增加 CNMAE 解析记录到云 WAF 提供的别名服务商，如图 3-16 所示。

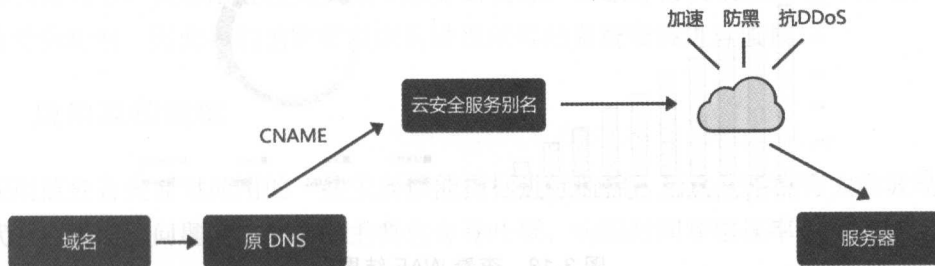


图 3-16 CNAME 方式接入云 WAF

- 如果站点是 HTTPS 的，那么需要上传对应的证书到云 WAF 平台上，如图 3-17 所示。



图 3-17 上传证书

- 配置云 WAF，一般云 WAF 使用默认策略即可。如果有特殊的需求，可以根据对应的产品手册进行自定义策略修改。
- 配置完成，查看相关的防护结果和报表，如图 3-18 和 3-19 所示。



图 3-18 查看 WAF 结果

域名:

时间 时间范围: 最近1小时 最近3小时 最近6小时 今天 昨天 最近3天 自定义 搜索

2017-04-24 15:48:50 ~ 2017-04-24 16:48:50 提交

源IP: User Agent: Referer:

Cookie: X-Forward-For: 服务器响应状态码:

| 访问时间 | 源IP | 访问域名 | 请求内容 | 请求主要头部字段 | 防护状态 | 响应状态 |
|----------------------|----------|-----------|----------------|--|------|------|
| 2017.4.4 12:12:22 | 23.3.3.3 | www.a.com | GET /HTTP/1.1 | cookie:xxx ua:xxx referer:xxx x-forward-for:xxx | 正常 | 200 |
| 2017.4.4 12:12:26 | 23.3.3.3 | www.a.com | POST /HTTP/1.1 | cookie:xxx ua:xxx referer:xxx x-forward-for:xxx | 已拦截 | 404 |

图 3-19 查看 WAF 报表

3.3 应用监控

应用经过努力，开发完成所有的功能并成功上线，但上线成功是不是终点？显然不是，我们的应用需要长期稳定地服务客户，而这就需要监控系统的支持。监控系统可以对关键资源（比如服务）或者应用数据进行持续的监控，并且在监控对象发生异常的时候及时发送报警，方便开发人员或运维人员快速处理问题，保证应用稳定运行。

在初创阶段，团队的主要精力应该还是以产品功能为主，而应用监控的主要目的是保证我们的产品能够正常服务用户，如果出现故障，能够及时发现并恢复。应用监控以应用进程的状态为主，但应用进程又受限于服务器资源，当服务器的状态出现异常时，应用进程也会受到影响，因此我们也需要对应用进程依赖的系统资源进行监控。

3.3.1 应用监控指标

应用监控首先要对应用的一些主要性能指标进行监控，当这些指标有大的波动时，一般都代表应用出现问题，这些指标主要包含吞吐量、响应时间和错误率。

吞吐量（Throughput）

每秒钟系统能够处理的请求数、任务数等。吞吐量的指标受到响应时间、服务器软硬件配置、网络状态等多方面因素影响。服务器硬件配置越高，吞吐量越大。网络越差，吞吐量越小。在低吞吐量下的响应时间的均值、分布比较稳定，不会产生太大的波动。在高吞吐量下，响应时间会随着吞吐量的增长而增长，增长的趋势可能是线性的，也可能接近指数。当吞吐量接近系统的峰值时，响应时间会出现激增。

响应时间（Responsiveness/Latency）

系统处理一个请求或者任务的耗时。响应时间的指标取决于具体的服务，如智能提示一类的服务，返回的数据有效周期短（用户多输入一个字母就需要重新请求），对实时性要求比较高。而导航一类的服务，由于返回结果的使用周期比较长（整个导航过程中），响应时间也就比较长。对于响应时间的统计，应从均值、.90 值、.99 值、分布等多个角度统计，而不仅仅是给出均值。

错误率 (Error Rate)

一批请求处理结果中出错请求所占比例，比如 HTTP 请求中 4XX, 5XX 请求所占的比例。错误率和服务的具体实现及环境相关，比如在移动网络下，可能因为运营商或者网络类型导致网络错误。

3.3.2 应用进程监控

应用进程监控的目标是确定我们的服务是否运行正常，最主要是要确定我们的应用进程是一直运行的，如果应用进程因为异常终止了，也就谈不上提供服务了。在一些情况下，应用进程虽然处于运行状态下，但无法正常对外提供服务，这也是我们监控的重点。

存活状态监控

应用进程可能会因为各种原因停止，比如机器资源不足（如 OOM）、内存非法访问等，在这种情况下，监控系统需要及时发现进程的存活状态，以在进程退出时可以快速自动拉起或者人工介入处理。

比如可以通过脚本定时检测的方式来检测目标进程是否存活，如果进程不存在，就自动拉起，也可以通过系统或者一些第三方提供的工具来检测，如 Monit。

进程资源监控

每个进程都会消耗一定的系统资源，除了系统整体的资源监控，我们必须对单个应用监控的 CPU、内存资源进行监控。如果操作系统整体资源使用率不高，但进程 CPU 或者内存使用异常，那么一般都说明应用进程有问题。

运行状态监控

除了进程的系统资源监控，我们还必须了解进程的运行状态。进程的运行状态可能跟具体的进程类型有关系，比如对 Java 进程，我们可能需要关心垃圾回收、内存使用、线程状态等；对于缓存来说，可能需要监控缓存的命中情况；对于数据库来说，我们可能需要监控慢查询的情况等。

Java 进程（JVM）监控

● GC（Garbage Collection）监控

GC 会严重影响应用的吞吐量和延时，目前主流的 JVM 都提供了比较方便的工具来观察 JVM 的性能。一般的 GC（Young GC，Full GC）数据都会包含 GC 的耗时、次数、GC 发生前的内存占用、GC 发生后的内存占用等数据，这些数据可以帮助你诊断 GC 是否已经影响了应用的性能。

● 内存监控

内存配置不正确可能会影响 GC 性能，也可能因为内存不足导致 OOM，所以需要对应 JVM 的内存（堆内存和非堆内存）使用进行监控，以便及时根据内存使用的趋势调整 JVM 配置。

● 线程监控

线程在运行过程中可能会因为锁等资源问题处理挂起或者死锁状态，如果没有线程相关监控，就不能及时发现线程导致的问题。线程监控主要监控 JVM 中存活的线程数量，以及各个线程的状态，及时对线程的状态进行评估，如果发现大量线程挂起或者线程死锁，可以快速产生报警。

3.3.3 操作系统监控

一般情况下，通过应用进程状态我们可以大致判断应用是否正常，如果在某些场景下，单纯依赖应用进程数据无法判断时，就需要依赖操作系统的一些监控数据来做参考。

操作系统提供对运行硬件资源的封装，从互联网应用的角度看，一般需要关注如下子系统的基础监控：CPU、内存、磁盘及网络等。这几个子系统之间相互影响，一个子系统出问题可能会影响其他几个子系统正常工作。

CPU

CPU 使用率情况和使用它的资源密切相关。系统内核调度器的主要职责是调度两种不同的资源：线程及中断。调度器针对不同的资源有不同的优先级。一般来说，优先级从高到低依次是：中断处理、内核线程和普通用户态进程（不考虑实时进程）。中断由硬件设备产生，设备在完成一项任务时，比如网上传输完一个数据包或者磁盘读完数据等，就会产

生一个中断，操作系统在接收到中断后，需要调用对应的中断处理过程。所有内核相关的处理过程（比如系统调用），中断处理以及内核线程都运行在内核态。除此之外的，一般我们所熟悉的 Web 服务相关的程序（Web 服务器、数据库服务器及各种中间件程序）都属于普通用户态进程，它们通过系统调用访问内核资源。为了正确理解或者能够通过操作系统所提供的信息判断应用是否正常，我们需要理解一些概念。

● 上下文切换（Context Switch）

现代的单核 CPU 在同一时间也只能运行一个线程，多个 CPU 核心可以同时运行多个线程。但一个标准的 Linux 内核，却可以同时（用户角度）运行几十到几万的线程。在只有一个 CPU 核心的情况下，内核就需要在这些线程之间进行调度，以保证每个任务都能得到 CPU 时间。每个线程都会分配一个时间片，在这个时间片用户有更高优先级的任务到达时，当前线程的 CPU 资源就会被抢占，然后放到一个等待队列，这就称为上下文切换。上下文切换的频率越高，说明内核需要花更多的时间做任务调度，而不是实际的运行任务。

● 运行队列（Run Queue）

每个 CPU 核心都要维护一个运行队列，保存当前可运行但暂未执行的线程。理想情况下，CPU 应该一直在运行或者执行线程，但是如果 CPU 的使用率比较高，有些线程就会处理等待状态，保存在运行队列中。运行队列越大，线程等待运行的时间就越长。一般通过 Load 来描述运行队列的状态。系统 Load 描述的是当前系统中正在执行的线程与正在运行队列中等待线程的综合情况。

● CPU 使用率

CPU 使用率反映的是 CPU 的使用情况，一般分如下几类：用户时间（User Time），CPU 执行用户空间线程的时间；系统时间（System Time），CPU 执行内核线程或者中断的时间；Wait IO（iowait）时间，CPU 因为所有线程都在等待 I/O 请求完成或者处理 idle 状态的时间；空闲时间（Idle），CPU 处理完全空闲状态的时间。

● CPU 负载判断

综合前面提到的几个方面（运行队列、上下文切换和 CPU 使用率）的数据就可以对当前的 CPU 负载做一个大致的判断。一般的判断标准如下。

1. 如果 CPU 满负载运行，那么 65%~70% 的用户时间，30%~35% 的系统时间，0%~

5%的空间时间是比较正常的。

2. 单个 CPU 核心的运行队列一般不能超过 1~3 个运行线程，也就是说一般系统 Load 不能超过系统核心数的 2~3 倍。
3. 上下文切换的频率直接跟 CPU 使用率有关，如果 CPU 使用率比较高，而且比较符合前面提到的用户系统时间分布，那么就算上下文切换比较多也可以接受。

在 Linux 下，一般可以通过 `vmstat` 命令来查看 CPU 的负载情况。

```
$ vmstat 1
procs -----memory----- ---swap-- -----io----- -system--
-----cpu-----
  r b  swpd  free  buff  cache  si  so  bi  bo  in  cs us sy id wa l
0   0 148868 86376 1297956 0 0 0 22 1 2 1 3 95 0
  1 0      0 149136 86384 1297980 0 0 0 12 3082 11625 2 5 93 1
  0 0      0 148548 86384 1297988 0 0 0 0 2383 10122 2 3 95 0
  0 0      0 148756 86384 1297988 0 0 0 4 1837 9491 1 2 97 0
  0 0      0 148808 86384 1297988 0 0 0 0 2729 10232 1 2 97 0
```

`vmstat` 的输出对上下文切换的情况 (cs)、运行队列的情况 (r) 及 CPU 使用情况都有统计。如果是多核系统，就可以通过 `mpstat` 命令查看 CPU 负载情况。

```
$ mpstat -P ALL 1
Linux 3.18.20-nce-amd64 (perf-tools-22177) 12/18/16 _x86_64_
(2 CPU)
06:17:49 CPU %usr %nice %sys %iowait %irq %soft %steal
%guest %idle
06:17:50 all 0.51 0.00 1.52 0.00 0.00 0.00 0.00
0.00 97.97
06:17:50 0 1.02 0.00 2.04 0.00 0.00 0.00 0.00
0.00 96.94
06:17:50 1 1.00 0.00 1.00 0.00 0.00 0.00 0.00
0.00 98.00
06:17:50 CPU %usr %nice %sys %iowait %irq %soft %steal
%guest %idle
06:17:51 all 0.51 0.00 0.51 0.00 0.00 0.00 0.00
```

```

0.00 98.98
      06:17:51      0  0.00  0.00  1.02  0.00  0.00  0.00  0.00
0.00 98.98
      06:17:51      1  0.00  0.00  0.00  0.00  0.00  0.00  0.00
0.00 100.00

```

内存

这里的“内存”包括物理内存和虚拟内存，虚拟内存（Virtual Memory）把计算机的内存空间扩展到硬盘，物理内存（RAM）和硬盘的一部分空间（Swap）组合在一起作为虚拟内存为进程提供了一个连贯的虚拟内存空间，好处是我们拥有的内存“变多了”，可以运行更多、更大的程序，坏处是把部分硬盘当内存用，整体性能受到影响，硬盘读写速度要比内存慢几个数量级，并且 RAM 和 Swap 之间的交换增加了系统的负担。

初期的内存监控主要关注两个点，一个是内存占用总量，一个是 Swap 空间的使用率。Linux 系统下可使用 `vmstat` 命令查看内存使用情况（如 `free`、`buff`、`cache` 等）。

```

$ vmstat 1

procs  -----memory-----  ---swap--  -----io-----  -system--
-----cpu---
   r  b  swpd  free  buff  cache  si  so  bi  bo  in  cs us sy id wa  1
0    0 148868 86376 1297956   0   0   0  22   1   2  1  3 95  0
   1  0    0 149136 86384 1297980   0   0   0  12 3082 11625  2  5 93  1
   0  0    0 148548 86384 1297988   0   0   0   0 2383 10122  2  3 95  0
   0  0    0 148756 86384 1297988   0   0   0   4 1837 9491  1  2 97  0
   0  0    0 148808 86384 1297988   0   0   0   0 2729 10232  1  2 97  0

```

内存使用量比较大，一方面可能是应用本身对内存需求比较大，那就需要扩容；另一方面可能是程序本身的内存泄露，导致内存使用量越来越大。

当系统的物理内存使用量低到一定程度时，系统就会开始使用 Swap 空间。如果观察到系统在持续的使用 Swap 空间，一般说明系统内存不够，不能满足应用的需求。

磁盘

磁盘 I/O 是操作系统中最慢的一个子系统，这主要是因为一般磁盘操作需要物理工作（旋转和寻道，当然 SSD 不需要）。所以，我们在设计应用系统时，都要考虑尽量减少磁盘

I/O 操作。一般的数据库系统，带持久化的队列或者缓存系统都需要写磁盘，我们需要对磁盘的状态进行监控。

● 磁盘使用空间

磁盘使用空间是最基本的磁盘监控指标，如果磁盘已经没有空闲空间，就会影响应用程序的运行。虽然有些程序会根据磁盘使用空间情况进行一定的保护措施（如基于磁盘使用空间进行流控），但大部分时候我们需要主动监控这个指标，发现磁盘使用空间比较高时，就要人工介入清理或者通过自动化工具定时清理。Linux 可通过 `df` 命令来查看磁盘空间的使用情况。

```
$ df -lah
Filesystem      Size  Used Avail Use% Mounted on
rootfs          20G  691M   18G   4% /proc
overlay         20G  691M   18G   4% /proc
tmpfs           1005M    0 1005M   0% /dev
```

● 磁盘使用率

磁盘使用率反映磁盘的繁忙程度，主要有以下几个指标。

1. IOPS (I/O Operations Per Second): 每秒钟完成的 I/O 操作数量。一般硬盘的最大 IOPS 都可以根据磁盘的转速，寻道时间等估计出来，比如 SATA 磁盘的 IOPS 大概在 80 左右，SAS 磁盘的 IOPS 大概在 150~200 左右，而 SSD 的 IOPS 可以达到几万。通过 IOPS 可以大概判断磁盘是否已经到瓶颈。
2. 平均等待时间 (await): 平均等待时间反映的是一个 I/O 请求从发出到磁盘完成处理所经过的时间，包含在磁盘 I/O 请求队列中排队的时间以及磁盘处理请求的时间，通过等待时间可以判断磁盘性能是否能满足应用对延迟的需求。如果等待时间增长得比较快，对应用的影响就会比较大。
3. 使用率 (ioutil): 在随机 I/O 比较多的多磁盘系统中，这个不是很精确，只有参考意义。但是如果 ioutil 一直增长，说明对应磁盘的利用率比较高。一般情况下，应该把使用率与平均等待时间放在一起来评估磁盘状态，如果使用率达到 100%，平均等待时间并没有明显增加，就说明磁盘还没有到瓶颈，但如果平均等待时间显著增加，就说明磁盘已经到瓶颈。

Linux 系统下可以通过 `iostat` 命令查看磁盘的负载情况。

```
$ iostat -x 1
Linux 3.18.20-nce-amd64 (perf-tools-22177)      12/18/16      _x86_64_      (2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.23    0.02   3.23    0.02    0.52   94.99

Device:            rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s  avgrq-sz  avgqu-sz   await  r_await  w_await  svctm  %uti
l
vda                 0.00     0.12    0.03    7.49     0.35    42.14    11.30     0.00     0.35    0.88    0.34    0.11    0.0
s
vdb                 0.00     0.00    0.00    0.00     0.00     0.00     7.05     0.00     0.09    0.09    0.00    0.09    0.0
0
vdc                 0.00     0.00    0.00    0.04     0.03     0.62    30.77     0.00    21.41    6.62    21.94    0.70    0.0
0

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.50    0.00   3.00    0.00    0.50   95.00

Device:            rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s  avgrq-sz  avgqu-sz   await  r_await  w_await  svctm  %uti
l
vda                 0.00     0.00    0.00   34.00     0.00   156.00     9.18     0.00     0.00    0.00    0.00    0.00    0.0
0
vdb                 0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00    0.0
0
vdc                 0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00    0.00
```

网络

互联网产品最主要的特征就是通过网络向用户提供服务，网络子系统的问题会影响服务的质量。

● 网络带宽

网络带宽是指在单位时间（一般指的是 1s）内传输的数据量。网络和高速公路类似，带宽越大，就类似高速公路的车道越多，其通行能力越强。

Linux 系统下，`ethtool` 命令可以查看网卡信息，包括网上的处理能力。

```
# ethtool eth0
Settings for eth0:

...
Speed: 10000Mb/s
Duplex: Full
...
```


● 网络流量（吞吐量）

网络流量反映单位时间内经过网卡的数据量，一般能够反映业务的负载。如果流量突然飙升，有可能说明用户请求量变多，也有可能是碰到网络攻击。如果流量突然降低，在确定用户负载波动不大的情况下，说明 Web 服务器可能出现故障，无法正常接收用户请求。sar 命令可以提供网上的流量统计信息。

```
$ sar -n DEV 1
Linux 3.2.0-4-amd64 (classa-apml2.server.163.org) 12/18/2016
_x86_64_ (4 CPU)
02:31:03 PM IFACE rxpck/s txpck/s rxkB/s txkB/s rxcmp/s
txcmp/s rxmcs/s
02:31:04 PM lo 0.00 0.00 0.00 0.00 0.00
0.00 0.00
02:31:04 PM eth1 8754.00 9126.00 2455.46 4088.72 0.00
0.00 0.00
02:31:04 PM eth0 122.00 156.00 25.47 14.77 0.00
0.00 0.00
02:31:04 PM IFACE rxpck/s txpck/s rxkB/s txkB/s rxcmp/s
txcmp/s rxmcs/s
02:31:05 PM lo 0.00 0.00 0.00 0.00 0.00
0.00 0.00
02:31:05 PM eth1 7896.00 8257.00 2249.59 3934.42 0.00
0.00 0.00
02:31:05 PM eth0 180.00 211.00 34.04 20.61 0.00
0.00 0.00
```

● 连接数量

连接数量是客户端与服务端连接数量的统计，连接数量有比较大的波动时，一般也预示着应用程序的问题。连接数量上经常会遇到的两个问题：一个是因为同时打开文件数目的限制（可调整）导致无法建立新的连接；另一个是端口范围太小无法分配端口导致无法建立新的连接。

● 连接状态

一般是指 TCP 协议中的连接状态，主要关注 ESTABLISHED、CLOSE_WAIT、

TIME_WAIT 这几个状态：比如是否存在大量 TIME_WAIT、CLOSE_WAIT 状态的连接，ESTABLISHED 连接的数量是否有大的波动等。连接状态及其数量的统计，在 Linux 系统下可以通过 ss 命令来完成。

```
$ ss -s
Total: 160 (kernel 0)
TCP: 99325 (estab 38, closed 99271, orphaned 0, synrecv 0, timewait
99269/0), ports 0 ...
```

小 结

本章我们分析了一个应用在初创期，从最初的技术选型到上线运行的全过程。在技术选型介绍中，我们分析了做业务框架选型时要考虑的因素，并以 Java 做一个 Web 系统为例进行介绍。同时也介绍了关于数据库、缓存及图片视频这些非结构化数据存储的技术选型。在架构实践中，我们主要关注了如何实现快速迭代开发及交付部署，高可用及应用安全等问题。最后，当系统上线后，我们还需要对运行的系统进行监控，主要介绍了一些监控的指标，以期系统遇到问题时，能被我们先行发现和快速解决。在初创期，我们的目标是使用工程化的方式，实现一个架构系统的成长，为接下来满足业务膨胀对系统扩展及增强的需求提供良好的基础。

第 4 章 快速成长期应用架构实践

在经过最初的业务原型验证和上线运行期之后，用户业务进入了高速成长阶段。在这一阶段，业务重点不再是方向上的调整，而是在原来基础上的不断深挖、扩展；开发不仅是功能的实现，还需要兼顾成本和性能；系统不再是单体架构，还会涉及系统的扩展和多系统之间的通信；高可用也不仅是服务自动拉起或者并行扩展，还需要考虑数据可靠、对用户影响，以及服务等级协议（SLA）。

本章我们将以上述挑战为出发点，介绍如何通过引入新的工具、新的架构，对原有系统进行升级和优化，来更好满足这一阶段需求，并为产品的进一步发展打下基础。

4.1 关键业务需求

随着用户业务的发展，原来的功能已经无法满足要求，需要增强或者增加新的功能。在用户数和访问量达到一定规模后，原先单体架构下的简单功能，如计数和排序，将变得复杂；随着业务深入，定期举行的秒杀、促销等活动，给系统带了巨大的压力；由于数据量的飞速增长，单纯的数据库或者内存检索已经无法满足不断增加的各种查询需求；随着业务数据量的增加，产品价值的提高，如何收集系统运行数据，分析业务运行状态也成了基本需求。接下来我们聚焦这一阶段的关键业务需求，并给出相应的解决方案。

4.1.1 计数与排序

在单体架构下，通过简单的内存数据和对应算法就可以实现计数和排序功能。但是在大量数据和多节点协作的环境下，基于单点内存操作的实现会遇到高并发、数据同步、实时获取等问题。在这一阶段，通用方法是使用 Redis 的原生命令来实现计数和排序。

计数

在 Redis 中可用于计数的对象有字符串（string）、哈希表（hash）和有序集合（zset）3 种，对应的命令分别是 incr/incrby、hincrby 和 zincrby。

网站可以从用户的访问、交互中收集到有价值的信息。通过记录各个页面的被访问次数，我们可以根据基本的访问计数信息来决定如何缓存页面，从而减少页面载入时间并提升页面的响应速度，优化用户体验。

计数器

要实现网页点击量统计，需要设计一个时间序列计数器，对网页的点击量按不同的时间精度（1s、5s、1min、5min、1h、5h、1d 等）计数，以对网站和网页监视和分析。

数据建模以网页的地址作为 KEY，定义一个有序集合（zset），内部各成员（member）分别由计数器的精度和计数器的名字组成，所有成员的分值（score）都是 0。这样所有精度的计数器都保存在了这个有序集合里，不包含任何重复元素，并且能够允许一个接一个地遍历所有元素。

对于每个计数器及每种精度，如网页的点击量计数器和 5s，设计使用一个哈希表（hash）对象来存储网页在每 5s 时间片之内获得的点击量。其中，哈希表的每个原生的 field 都是某个时间片的开始时间，而原生的 field 对应的值则存储了网页在该时间片内获得的点击量。如图 4-1 所示。

● 更新计数器信息示例代码。

```
PRECESION = [1, 5, 60, 300, 3600, 18000, 86400] def update_counter(conn,
name, count=1, now=None):
    ----now = now or time.time()
    ----pipe = conn.pipeline() ----for prec in PRECISION:
    -----pnow = int(now / prec) * prec
    -----hash = '%s:%s' % (prec, name)
    -----pipe.zadd('http://.....xxxxxx', hash, 0)
    -----pipe.hincrby('count:' + hash, pnow, count)
    ----pipe.execute()
```

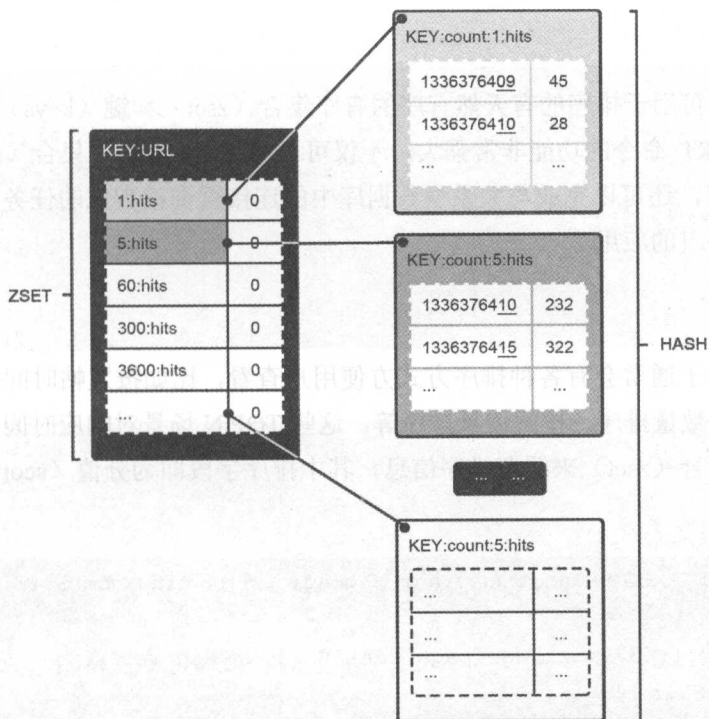


图 4-1 网页点击计数器的实现

- 获取计数器信息示例代码。

```
def get_counter(conn, name, precision):
    ----hash = '%s:%s' % (precision, name)
    ----data = conn.hgetall('count:' + hash)
    ----to_return = []
    ----for key, value in data.iteritems():
    -----to_return.append((int(key), int(value)))
    ----to_return.sort()
    ----return to_return;
```

当然，这里只介绍了网页点击量数据的存储模型，如果我们一味地对计数器进行更新而不执行任何清理操作的话，那么程序最终将会因为存储了过多的数据而导致内存不足，由于我们事先已经将所有已知的计数器都记录到一个有序集合里面，所以对计数器进行清理只需要遍历这个有序集合，并删除其中的旧计数器即可。

排序

在 Redis 中可用于排序的有天然有序的有序集合（zset）和键（keys）类型中的 SORT 命令，其中 SORT 命令的功能非常强大，不仅可以对列表（list）、集合（set）和有序集合（zset）进行排序，还可以完成与关系型数据库中的连接查询相类似的任务，下面分别以两个例子来介绍各自的应用。

帖子排序

论坛中的帖子通常会有各种排序方式方便用户查看，比如按发帖时间排序、按回复时间排序、按回复数量排序、按阅读量排序等，这些 TOP N 场景对响应时间要求比较高，非常适宜用有序集合（zset）来缓存排序信息，其中排序字段即为分值（score）字段。

● 例子

```
127.0.0.1:6379> zadd page_rank 10 google.com 8 bing.com 6 163.com 9 baidu.com
(integer) 4
127.0.0.1:6379> zrange page_rank 0 -1 withscores
1) "163.com"
2) "6"
3) "bing.com"
4) "8"
5) "baidu.com"
6) "9"
7) "google.com"
8) "10"
```

SORT 命令

```
SORT key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]]
[ASC | DESC] [ALPHA] [STORE destination]
```

SORT 命令提供了多种参数，可以对列表，集合和有序集合进行排序，此外还可以根据降序升序来对元素进行排序（DESC、ASC）；将元素看作是数字还是二进制字符串来进行排序（ALPHA）；使用排序元素之外的其他值作为权重来进行排序（BY pattern）。

下面代码清单展示了 SORT 命令的具体功能使用。

● 对列表（list）进行排序

1. 顺序

```
127.0.0.1:6379> lpush mylist 30 10 8 19
(integer) 4
127.0.0.1:6379> sort price
1) "8"
2) "10"
3) "19"
4) "30"
```

2. 逆序

```
127.0.0.1:6379> sort price desc
1) "30"
2) "19"
3) "10"
4) "8"
```

3. 使用 alpha 修饰符对字符串进行排序

```
127.0.0.1:6379> lpush website www.163.com www.kaola.com www.baidu.com
(integer) 3
127.0.0.1:6379> sort website alpha
1) "www.163.com"
2) "www.baidu.com"
3) "www.kaola.com"
```

● 使用 limit 修饰符限制返回结果

```
127.0.0.1:6379> rpush num 1 4 2 7 9 6 5 3 8 10
(integer) 10
127.0.0.1:6379> sort num limit 0 5
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
```

● 使用外部 key 进行排序

可以使用外部 key 的数据作为权重，代替默认的直接对比键值的方式来进行排序。假设现在有用户数据如表 4-1 所示。

表 4-1 用户数据示例

| uid | user_name_{uid} | user_level_{uid} |
|-----|-----------------|------------------|
| 1 | helifu | 888 |
| 2 | netease | 666 |
| 3 | kaola | 777 |
| 4 | ncr | 4444 |

以下将哈希表 (hash) 作为 by 和 get 的参数，by 和 get 选项都可以用 key->field 的格式来获取哈希表中的域的值，其中 key 表示哈希表键，而 field 则表示哈希表的域。

1. 数据输入到 Redis 中

```
127.0.0.1:6379> hmset user_info_1 name helifu level 888
OK
127.0.0.1:6379> hmset user_info_2 name netease level 666
OK
127.0.0.1:6379> hmset user_info_3 name kaola level 777
OK
127.0.0.1:6379> hmset user_info_4 name ncr level 4444
OK
```

2. by 选项

通过使用 by 选项，让 uid 按其他键的元素来排序。

例如以下代码让 uid 键按照 user_info_*->level 的大小来排序。

```
127.0.0.1:6379> sort uid by user_info_*->level
1) "2"
2) "3"
3) "1"
4) "4"
```

3. get 选项

使用 get 选项，可以根据排序的结果来取出相应的键值。

例如以下代码先让 uid 键按照 user_info_*->level 的大小来排序,然后再取出 user_info_*->name 的值。

```
127.0.0.1:6379> sort uid by user_info_*->level get user_info_*->name
1) "netease"
2) "kaola"
3) "helifu"
4) "ncr"
```

现在的排序结果要比只使用 by 选项要直观得多。

4. 排序获取多个外部 key

可以同时使用多个 get 选项,获取多个外部键的值。

```
127.0.0.1:6379> sort uid get # get user_info_*->level get
user_info_*->name
1) "1"
2) "888"
3) "helifu"
4) "2"
5) "666"
6) "netease"
7) "3"
8) "777"
9) "kaola"
10) "4"
11) "4444"
12) "ncr"
```

5. 不排序获取多个外部 key

```
127.0.0.1:6379> sort uid by not-exists-key get # get user_info_*->level
get user_info_*->name
1) "4"
2) "4444"
3) "nc"
4) "3"
```

```
5) "777"  
6) "kaola"  
7) "2"  
8) "666"  
9) "netease"  
10) "1"  
11) "888"  
12) "helifu"
```

● 保存排序结果

```
127.0.0.1:6379> lrange old 0 -1  
1) "1"  
2) "3"  
3) "5"  
6) "2"  
7) "4"  
  
127.0.0.1:6379> sort old store new  
(integer) 5  
127.0.0.1:6379> type new list  
127.0.0.1:6379> lrange new 0 -1  
1) "1"  
2) "2"  
3) "3"  
4) "4"  
5) "5"
```

`SORT` 命令的时间复杂度用公式表示为 $O(N+M*\log(M))$ ，其中 N 为要排序的列表或集合内的元素数量， M 为要返回的元素数量。如果只是使用 `SORT` 命令的 `get` 选项获取数据而没有进行排序，时间复杂度为 $O(N)$ 。

云环境下的实践

在云服务中实现计数和排序，可以自己使用云主机搭建 Redis 服务，也可以使用云计算服务商提供的 Redis 服务。

对于高可用和性能有要求的场景，建议使用云计算服务商提供的 Redis 服务。专业的

服务商会从底层到应用本身进行良好的优化,可用率、性能指标也远高于自己搭建的 Redis 实例。同时,由于服务商提供了各种工具,开发运维成本也更低。

以网易云为例,网易云基础服务提供了名为 NCR (Netease Cloud Redis) 的缓存服务,兼容开源 Redis 协议。并根据用户具体使用需求和场景,提供了主从版本和分布式集群版本两种架构。

主从服务版

如图 4-2 所示,主从版本实例都提供一主一从两个 Redis 实例,分别部署在不同可用域的节点上,以确保服务安全可靠。在单点故障时,主从服务通过主备切换来实现高可用。

主从版本使用较低的成本提供了高可用服务,但是也存在无法并行扩展等问题,因此适合数据量有限、对高可用有要求的产品使用。



图 4-2 主从服务架构

分布式集群

分布式集群采用官方 Redis 集群方案, gossip/p2p 的无中心节点设计实现,无代理设计客户端直接与 Redis 集群的每个节点连接,计算出 Key 所在节点直接在对应的 Redis 节点上执行命令,如图 4-3 所示,详细的过程请参考后续 Redis Cluster 的相关介绍。

分布式集群采用多活模式,支持并行扩展,因此在性能、可用率方面有明显优势。但是由于分布式集群最少需要 3 个节点,因此成本会较高,适合对可用率、性能有较高要求

的用户使用。

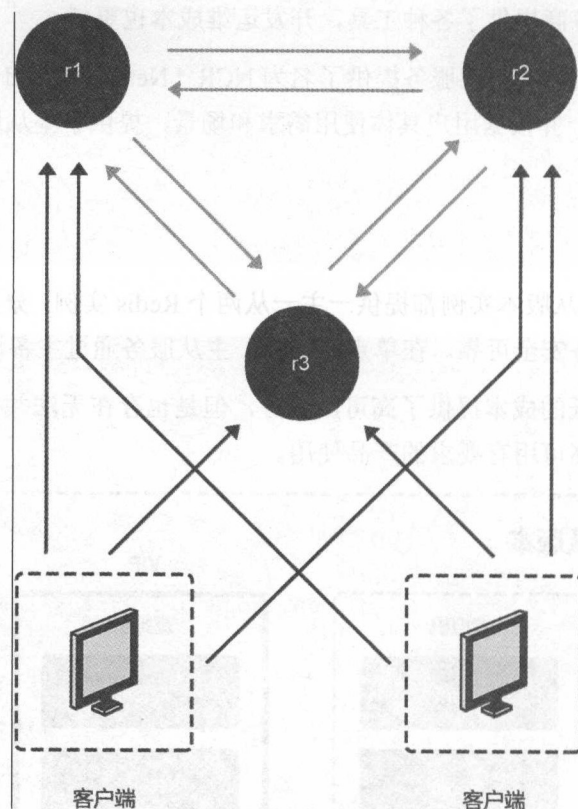


图 4-3 分布式集群架构

4.1.2 秒杀

把秒杀服务单列出来进行分析，主要有下面两个原因。

1. 秒杀服务的重要性：秒杀活动本身已经是很多业务推广的重要方式之一，大部分的电商类业务都会涉及这一促销方式。很多非直接秒杀的业务（如火车票购票），在实际运行时也会碰到类似秒杀的场景。秒杀实际上就是在瞬时极大并发场景下如何保证系统正常运行的问题，而这种场景对很多系统都是无法避免的，因此在系统设计时，我们往往要考虑到秒杀的影响。
2. 系统实现难度：秒杀最能考验系统负载能力，瞬间涌入平时数十倍甚至数百倍的压

力，对开发和运维人员来说都是噩梦，这也为系统设计带来了巨大的挑战。针对秒杀活动的处理，是一个系统性的设计，并不是单一模块或者层面可以解决的问题，需要从系统设计整体进行考量。

处理秒杀的指导思路

秒杀的核心问题就是极高并发处理，由于系统要在瞬时承受平时数十倍甚至上百倍的流量，这往往超出系统上限，因此处理秒杀的核心思路是流控和性能优化。

● 流控

1. 请求流控

尽可能在上游拦截和限制请求，限制流入后端的量，保证后端系统正常。

因为无论多少人参与秒杀，实际成交往往是有限的，而且远小于参加秒杀的人数，因此可以通过前端系统进行拦截，限制最终流入系统的请求数量，来保证系统正常进行。

2. 客户端流控

在客户端进行访问限制，较为合适的做法是屏蔽用户高频请求，比如在网页中设置 5s 一次访问限制，可以防止用户过度刷接口。这种做法较为简单，用户体验也尚可，可以拦截大部分小白用户的异常访问，比如狂刷 F5。关键是要明确告知用户，如果像一些抢购系统那样假装提交一个排队页面但又不回应任何请求，就是赤裸裸的欺骗了。

3. Web 端流控

对客户端，特别是页面端的限流，对稍有编程知识或者网络基础的用户而言没有作用（可以简单修改 JS 或者模拟请求），因此服务端流控是必要的。服务端限流的配置方法有很多种，现在的主流 Web 服务器一般都支持配置访问限制，可以通过配置实现简单的流控。

但是这种限制一般都在协议层。如果要实现更为精细的访问限制（根据业务逻辑限流），可以在后端服务器上，对不同业务实现访问限制。常见做法是可以通过在内存或缓存服务中加入请求访问信息，来实现访问量限制。

4. 后端系统流控

上述的流控做法只能限制用户异常访问，如果正常访问的用户数量很多，就有后端系统压力过大甚至异常宕机的可能，因此需要后端系统流量控制。

对于后端系统的访问限制可以通过异步处理、消息队列、并发限制等方式实现。核心思路是保证后端系统的压力维持在可以正常处理的水平。对于超过系统负载的请求，可以选择直接拒绝，以此来对系统进行保护，保证在极限压力的情况下，系统有合理范围内的处理能力。

● 系统架构优化

除了流控之外，提高系统的处理能力也是非常重要的，通过系统设计和架构优化，可以提高系统的吞吐量和抗压能力。关于通用系统性能的提升，已经超出本节范围，这里只会提几点和秒杀相关的优化。

1. 读取加速：在秒杀活动中，数据需求一般都是读多写少。20 万人抢 2000 个商品，最后提交的订单最多也就 2000 个，但是在秒杀过程中，这 20 万人会一直产生大量的读取请求。因此可以使用缓存服务对用户请求进行缓存优化，把一些高频访问的内容放到缓存中去。对于更大规模的系统，可以通过静态文件分离、CDN 服务等把用户请求分散到外围设施中去，以此来分担系统压力。
2. 异步处理和排队：通过消息队列和异步调用的方式可以实现接口异步处理，快速响应用户请求，在后端有较为充足的时间来处理实际的用户操作，提高对用户请求的响应速度，从而提升用户体验。通过消息队列还可以隔离前端的压力，实现排队系统，在涌入大量压力的情况下保证系统可以按照正常速率来处理请求，不会被流量压垮。
3. 无状态服务设计：相对于有状态服务，无状态服务更容易进行扩展，实现无状态化的服务可以在秒杀活动前进行快速扩容。而云化的服务更是有着先天的扩容优势，一般都可以实现分钟级别的资源扩容。

● 系统扩容

这项内容是在云计算环境下才成为可能，相对于传统的 IT 行业，云计算提供了快速的系统交付能力（min VS. day），因此可以做到按需分配，在业务需要时实现资源的并行扩展。

对一次成功的秒杀活动来说，无论如何限流，如何优化系统，最终产生数倍于正常请求的压力是很正常的。因此临时性的系统扩容必不可少，系统扩容包括以下 3 个方面。

1. 增加系统规格：可以预先增加系统容量，比如提高系统带宽、购买更多流量等。

2. 服务扩展：无状态服务+负载均衡可以直接进行水平扩展，有状态的服务则需要进行较为复杂的垂直扩展，增大实例规格。
3. 后端系统扩容：缓存服务和数据库服务都可以进行容量扩展。

秒杀服务实践

一般来说，流控的实现，特别是业务层流控，依赖于业务自身的设计，因此云计算提供的服务在于更多、更完善的基础设计，来支持用户进行更简单的架构优化和扩容能力。

系统架构优化

通过 CDN 服务和对象存储服务来分离静态资源，实现静态资源的加速，避免服务器被大量静态资源请求过度占用。要实现异步的消息处理，可以使用队列服务来传输消息，以达到消息异步化和流控。

系统扩容

云服务会提供按需计费的资源分配方式和分钟级甚至秒级的资源交付能力，根据需要快速进行资源定制和交付。

内部系统可以通过负载均衡等服务实现并行扩展，在网易云基础服务中，用户可以直接使用 Kubernetes 的 Replication Controller 服务实现在线水平扩容。对于对外提供的 Web 系统，可以通过负载均衡服务实现水平在线扩展。

对于后端系统来说，建议使用云计算服务商提供的基础服务来实现并行扩展。例如，网易云基础服务就提供了分布式缓存服务和数据库服务，支持在线扩容。

4.1.3 全文检索

搜索，是用户获取信息的主要方式。在日常生活中，我们不管是购物（淘宝）、吃饭（大众点评、美团网）还是旅游（携程、去哪儿），都离不开搜索的应用。搜索几乎成为每个网站、APP 甚至是操作系统的标配。在用户面前，搜索通常只是展示为一个搜索框，非常干净简洁，但它背后的原理可没那么简单，一个框的背后包含的是一整套搜索引擎的原理。假如我们需要搭建一个搜索系统为用户提供服务，我们又需要了解什么呢？

1. 基本原理

首先，我们需要知道全文检索的基本原理，了解全文检索系统在实际应用中是如何工作的。

通常，在文本中查找一个内容时，我们会采取顺序查找的方法。譬如现在手头上有一本计算机书籍，我们需要查找出里面包含了“计算机”和“人工智能”关键字的章节。一种方法就是从头到尾阅读这本计算机书籍，在每个章节都留心是否包含了“计算机”和“人工智能”这两个词。这种线性扫描就是最简单的计算机文档检索方式。这个过程通常称为 **grepping**，它来自于 Unix 下的一个文本扫描命令 **grep**。在文本内进行 **grepping** 扫描很快，使用现代计算机会更快，并且在扫描过程中还可以通过使用正则表达式来支持通配符查找。总之，在使用现代计算机的条件下，对一个规模不大的文档集进行线性扫描非常简单，根本不需要做额外的处理。但是，很多情况下只采用上述扫描方式是远远不够的，我们需要做更多的处理。这些情况如下所述。

- 大规模文档集条件下的快速查找。用户的数据正在进行爆发性的增长，我们可能需要在几十亿到上万亿规模下的数据进行查找。
- 有时我们需要更灵活的匹配方式。比如，在 **grep** 命令下不能支持诸如“计算机 NEAR 人工智能”之类的查询，这里的 NEAR 操作符的定义可能为“5 个词之内”或者“同一句子中”。
- 需要对结果进行排序。很多情况下，用户希望在多个满足自己需求的文档中得到最佳答案。

此时，我们不能再采用上面的线性扫描方式。一种非线性扫描的方式是事先给文档建立索引 (**Index**)。回到上面所述的例子，假设我们让计算机对整书本预先做一遍处理，找出书中所有的单词都出现在了哪几个章节 (由于单词会重复，通常都不会呈现爆炸式增长)，并记录下来。此时再查找“计算机”和“人工智能”单词出现在哪几个章节中，只需要将保存了它们出现过的章节做合并等处理，即可快速寻找出结果。存储单词的数据结构在信息检索的专业术语中叫“倒排索引” (**Inverted Index**)，因为它和正向的从章节映射到单词关系相反，是倒着的索引映射关系。

这种先对整个文本建立索引，再根据索引在文本中进行查找的过程就是全文检索 (**Full-text Search**) 的过程。图 4-4 展示了全文检索的一般过程。

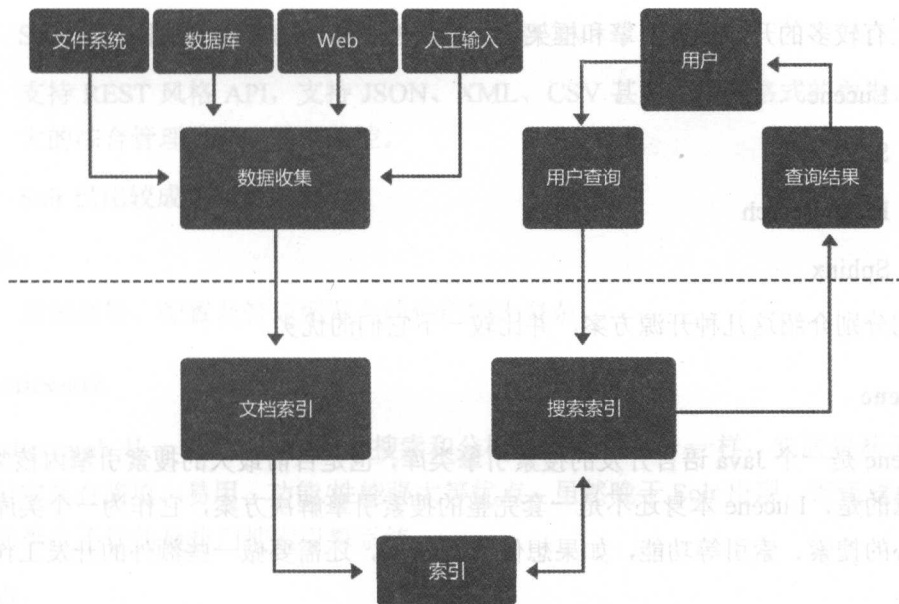


图 4-4 全文检索的一般过程

首先是数据收集的过程 (Gather Data)，数据可以来源于文件系统、数据库、Web 抓取甚至是用户输入。数据收集完成后我们对数据按上述的原理建立索引 (Index Documents) 并保存至 Index 索引库中。在图的右边，用户使用方面，我们在页面或 API 等获取到用户的搜索请求 (Get Users' Query)，并根据搜索请求对索引库进行查询 (Search Index)，然后对所有的结果进行打分、排序等操作，最终形成一个正确的结果返回给用户并展示 (Present Search Results)。当然在实现流程中还包含了数据抓取/爬虫、链接分析、分词、自然语言处理、索引结构、打分排序模型、文本分类、分布式搜索系统等技术，这是最简单抽象的流程描述。关于索引过程和搜索过程更详细的技术就不做更多介绍了，感兴趣的同学请参考其他专业书籍。

2. 开源框架

根据上面的描述我们知道了全文检索的基本原理，但要是想自己从头实现一套搜索系统还是很困难的，没有一个专业的团队、一定的时间基本上做出不来，而且系统实现之后还需要面临生产环境等各种问题的考验，研发和维护成本都无比巨大。不过，现代的程序开发环境早已今非昔比，开源思想深入人心，开源软件大量涌现。没有特殊的需求，没有人会重新开发一套软件。我们可以站在开源巨人的肩膀上，直接利用开源软件的优势。目

前市面上有较多的开源搜索引擎和框架，比较成熟和活跃的有以下几种。

- Lucene
- Solr
- Elasticsearch
- Sphinx

我们分别介绍这几种开源方案，并比较一下它们的优劣。

Lucene

Lucene 是一个 Java 语言开发的搜索引擎类库，也是目前最火的搜索引擎内核类库。但需要注意的是，Lucene 本身还不是一套完整的搜索引擎解决方案，它作为一个类库只是包含了核心的搜索、索引等功能，如果想使用 Lucene，还需要做一些额外的开发工作。

优点：

- Apache 顶级项目，仍在持续快速进步。
- 成熟的解决方案，有很多成功案例。
- 庞大而活跃的开发社区，大量的开发人员。
- 虽然它只是一个类库，但经过简单的定制和开发，就可以满足绝大部分常见的需求；经过优化，可以支持企业级别量级的搜索。

缺点：

- 需要额外的开发工作。系统的扩展、分布式、高可用性等特性都需要自己实现。

Solr

Solr 是基于 Lucene 开发、并由开发 Lucene 的同一帮人维护的一套全文搜索系统，Solr 的版本通常和 Lucene 一同发布。Solr 是最流行的企业级搜索引擎之一。

优点：

- 由于和 Lucene 共同维护，Solr 也有一个成熟、活跃的社区。
- Solr 支持高级搜索特性，比如短语搜索、通配符搜索、join、group 等。

- Solr 支持高吞吐流量、高度可扩展和故障恢复。
- 支持 REST 风格 API，支持 JSON、XML、CSV 甚至二进制格式的数据；功能强大的综合管理页面、易于监控。
- Solr 已比较成熟、稳定。

缺点：

- 系统部署、配置及管理配置上的使用较为复杂。

Elasticsearch

Elasticsearch 是一个实时的分布式搜索和分析引擎，和 Solr 一样，它底层基于 Lucene 框架。但它具有简单、易用、功能/性能强大等优点，虽然晚于 Solr 出现，但迅速成长，已成为目前当仁不让的最热门搜索引擎系统。

优点：

- 支持 REST 风格的 HTTP API，并且事件完全受 API 驱动，几乎所有的操作都可以通过使用 JSON 格式的 RESTful HTTP API 完成。
- 支持多租户/多索引（multi-tenancy），支持全面的高级搜索特性。
- 索引模式自由（Schema Free），支持 JSON 格式数据；部署、配置简单，便于使用。
- 强大的聚合（Aggregation）分析功能（取代 Lucene 传统的 Facets），便于用户对数据进行统计分析。

缺点：

- 几乎无缺点，在性能和资源上较 C++ 开发的 Sphinx 稍差。

Sphinx

Sphinx 是基于 C++ 开发的一个全文检索引擎，从最开始设计时就注重性能、搜索相关性及整合简单性等方面。Sphinx 可以批量索引和搜索 SQL 数据库、NoSQL 存储中的数据，并提供和 SQL 兼容的搜索查询接口。

优点：

- Sphinx 采用 C++ 开发，因此支持高速的构建索引及高性能的搜索。

- 支持 SQL/NoSQL 搜索。
- 方便的应用集成。
- 更高级的相似度计算排序模型。

缺点:

- 社区、生态等不如 Lucene 系发达, 可见的成功案例较少。

应用选型

通过上面的介绍, 相信大家对各个开源搜索系统所适用的场景有了一定了解。

如果是中小型企业, 想快速上手使用搜索功能, 可以选择 Elasticsearch 或 Solr。如果对搜索性能和节省资源要求比较苛刻, 可以考虑尝试 Sphinx。如果有很多定制化的搜索功能需求, 可以考虑在 Lucene 基础上做自定义开发。如果用于日志搜索, 或者有很多的统计、分析等需求, 可以选择 Elasticsearch。

3. 开源方案实践

如上述介绍, 在实际开发当中已有了很多现成的开源方案可供选择, 但我们还是需要额外再做一些事情。譬如集群搭建、维护和管理、高可用设计、故障恢复、最基本的机房申请及机器采购部署等。这也需要投入较高的人力和成本(虽然较自己研发已经节省很多), 并且还需要配备专业的搜索开发及运维人员。

而在网易云中, 我们基于开源 Elasticsearch 系统提供了一套简单的方案。我们把专业、复杂的搜索系统服务化和简单化, 并降低准入门槛和成本, 让用户可以直接使用平台化的搜索服务。我们提供了全面的近实时、高可用、高可靠、高扩展等搜索系统强大功能, 易于用户使用。用户使用网易云的云搜索后不再需要处理搜索系统的搭建与配置工作, 而只需要在云搜索服务的产品管理平台申请建立服务实例并配置索引数据格式, 申请完成后云搜索平台就会自动生成索引服务实例并提供全文检索服务。

4.1.4 日志收集

日志, 一组随时间增加的有序记录, 是开发人员最熟悉的一种数据。通常, 日志可以用来搜索查看关键状态、定位程序问题, 以及用于数据统计、分析等。

日志也是企业生产过程中产生的伟大财富。日志可以用来进行商业分析、用户行为判断和企业战略决策等，良好地利用日志可以产生巨大的价值。所以，日志的收集不管是在开发运维还是企业决策中都十分重要。

1. 第三方数据收集服务

在日志收集领域内，目前已经存在了种类繁多的日志收集工具，比较典型的有：`rsyslog`、`syslog-ng`、`Logstash`、`FacebookScribe`、`ClouderaFlume`、`Fluentd` 和 `GraylogCollector` 等。

`rsyslog` 是 `syslog` 的增强版，`Fedora`、`Ubuntu`、`RHEL6`、`CentOS6`、`Debian` 等诸多 Linux 发行版都已使用 `rsyslog` 替换 `syslog` 作为默认的日志系统。`rsyslog` 采用 C 语言实现，占用的资源少，性能高。专注于安全性及稳定性，适用于企业级别日志记录需求。`rsyslog` 可以传输 100 万+/s（日志条数）的数据到本地目的地。即使通过网络传输到远程目的地，也能达到几万至几十万条/每秒的级别。`rsyslog` 默认使用 `inotify` 实现文件监听（可以更改为 `polling` 模式，实时性较弱），实时收集日志数据。

`rsyslog` 支持实时监听日志文件、支持通配符匹配目录下的所有文件（支持输出通配符匹配的具体文件名）、支持记录文件读取位置、支持文件 `Rotated`、支持日志行首格式判断（多行合并成一行）、支持自定义 `tag`、支持直接输出至 `file/mysql/kafka/elasticsearch` 等、支持自定义日志输出模板，以及支持日志数据流控。

`syslog-ng` 具有开源、可伸缩和可扩展等特点，使用 `syslog-ng`，你可以从任何来源收集日志，以接近实时的处理，输出到各种各样的目的源。`syslog-ng` 灵活地收集、分析、分类和关联日志，存储和发送到日志分析工具。`syslog-ng` 支持常见的输入，支持 `BSDsyslog`（RFC3164）、RFC5424 协议、JSON 和 `journald` 消息格式。数据提取灵活，内置一组解析器，可以构建非常复杂事情。简化复杂的日志数据，`syslog-ng patterndb` 可以转化关联事件为一个统一格式。支持数据库存储，包括 SQL（MySQL，PostgreSQL，Oracle）、MongoDB 和 Redis。`syslog-ng` 支持消息队列，支持高级消息队列协议（AMQP）和面向简单的文本消息传递协议（STOMP）与更多的管道。`syslog-ng` 设计原则主要包括更好消息过滤粒度和更容易在不同防火墙网段转发信息。前者能够进行基于内容和优先权/facility 的过滤。后者支持主机链，即使日志消息经过了许多计算机的转发，也可以找出原发主机地址和整个转发链。

`Logstash` 是一个开源的服务器端数据处理管道，采集多种数据源的数据，转换后发送

到指定目的源。数据通常以各种格式分散在许多孤立系统上。Logstash 支持种类丰富的 inputs，以事件的方式从各种源获取输入，包括日志、Web 应用、数据存储设备和 AWS 服务等。在数据流从源到存储设备途中，Logstash 过滤事件，识别字段名，以便结构化数据，更有利于数据分析和创造商业价值。除了 Elasticsearch，Logstash 支持种类丰富的 outputs，可以把数据转发到合适的目的源。表 4-2 是几个典型产品的特性对比。

表 4-2 典型日志收集产品特性比较

| 名称 | 开发语言 | 性能 | 所占资源 | 支持 I/O 插件种类 | 社区活跃度 |
|----------------|------------------------|----|------|-------------|-------|
| rsyslog | C | 高 | 少 | 中 | 中 |
| syslog-ng | C | 高 | 少 | 中 | 中 |
| LogStash/Beats | LogStash:Ruby Beats:Go | 中 | 多 | 多 | 高 |
| FacebookScribe | C++ | 高 | 少 | 少 | 中 |
| ClouderaFlume | Java | 中 | 多 | 中 | 中 |
| Fluentd | Ruby | 中 | 多 | 多 | 高 |

2. 技术选型

由于日志收集的需求并非很复杂，此类工具大体上比较相似，用户只需要根据其特性选择合适自己需求的产品。

通常来说，对于日志收集客户端资源占用要求较高的，可以选择 C 语音开发的 rsyslog、syslog-ng。对于易用性要求较高，可以选择 Logstash、Beats。对于日志收集后接入的后端有特殊需求，可以参考 Fluentd 是否可以满足。如果公司用的是 Java 技术栈，可以选用 Cloudera Flume。

4.2 架构实践

除了基本的功能需求之外，一个互联网产品往往还有访问性能、高可用、可扩展等需要，这些统称为非功能需求。一般来说，功能需求往往可以通过开发业务模块来满足，而非功能需求往往要从系统架构设计出发，从基础上提供支持。

随着用户的增加，系统出现问题的影响也会增大。试想一下，一个小公司的主页，或者个人开发维护的一个 App 的无法访问，可能不会有多少关注。而支付宝、微信的宕机，

则会直接被推到新闻头条（2015 年支付宝光纤被挖路机挖断），并且会给用户带来严重的影响：鼓足勇气表白却发现信息丢失？掏出手机支付却发生支付失败，关键是还没带现金！在用户使用高峰时，一次故障就会给产品带来很大的伤害，如果频繁出现故障则基本等同于死刑判决。

同样，对一个小产品来说，偶发的延时、卡顿可能并不会有很大的影响（可能已经影响到了用户，只是范围、概率较小）。而对于一个较为成熟的产品，良好的性能则是影响产品生死存亡的基本问题。试想一下，如果支付宝、微信经常出现卡顿、变慢，甚至在访问高峰时崩溃，那它们还能支撑起现在的用户规模，甚至成为基础的服务设施吗？可以说，良好的访问性能，是一个产品从幼稚到成熟所必须解决的问题，也是一个成功产品的必备因素。实际上，很多有良好创意、商业前景很好的互联网产品，就是因无法满足用户增长带来的性能压力而夭折。

随着性能需求的不断增长，所需要考虑的因素越多，出问题的概率也越大。因此，用户数的不断增长带来的挑战和问题几乎呈几何倍数增加，如果没有良好的设计和规划，随着产品和业务的不断膨胀，我们往往会陷入“修改→引入新问题→继续调整→引入更多问题”的泥潭中无法自拔。

在这一阶段，架构设计的重点不再是业务本身功能实现和架构的构建，而是如何通过优化系统架构，来满足系统的高可用、并行扩展和系统加速等需要。

4.2.1 前端系统扩展

可扩展性是大规模系统稳定运行的基石。随着互联网用户的不断增加，一个成功产品的用户量往往是数以亿计，无论多强大的单点都无法满足这种规模的性能需求。因此系统的扩展是一个成功互联网产品的必然属性，无法进行扩展的产品，注定没有未来。

由于扩展性是一个非常大的范畴，并没有一个四海皆准的手段或者技术来实现，因此本节主要介绍较为通用的可扩展系统设计，并以网易云为例，来介绍基础设施对可扩展性的支持。

4.2.2 无状态服务设计

要实现系统的并行扩展，需要对原有的系统进行服务化拆分。在服务实现时，主要有两种实现方式，分别是无状态服务和有状态服务。

1. 特点

无状态服务

指的是服务在处理请求时，不依赖除了请求本身外的其他内容，也不会有除了响应请求之外的额外操作。如果要实现无状态服务的并行扩展，只需要对服务节点进行并行扩展，引入负载均衡即可。

有状态服务

指的是服务在处理一个请求时，除了请求自身的信息外，还需要依赖之前的请求处理结果。

对于有状态服务来说，服务本身的状态也是正确运行的一部分，因此有状态服务相对难以管理，无法通过简单地增加实例个数来实现并行扩展。

对比

从技术的角度来看，有状态服务和无状态服务只是服务的两种状态，本身并没有优劣之分。在实际的业务场景下，有状态服务和无状态服务相比，有各自的优势。

有状态服务的特性如下。

- 数据局部性：数据都在服务内部，不需要依赖外部数据服务强并发，有状态服务可以把状态信息都放在服务内部，在并发访问时不用考虑冲突等问题，而本地数据也可以提供更好的存取效率。因此，单个有状态服务可以提供更强的并发处理能力。
- 实现简单：可以把数据信息都放在本地，因此可以较少考虑多系统协同的问题，在开发时更为简单。

无状态服务的特性如下。

- 易扩展：可以通过加入负载均衡服务，增加实例个数进行简单的并行扩展易管理，由于不需要上下文信息，因此可以方便地管理，不需要考虑不同服务实例之间的差异。
- 易恢复：对于服务异常，不需要额外的状态信息，只需要重新拉起服务即可。而

在云计算环境下，可以直接建立新的服务实例，替代异常的服务节点即可。

总体来看，有状态服务在服务架构较为简单时，有易开发、高并发等优势，而无状态服务的优势则体现在服务管理、并行扩展方面。随着业务规模的扩大、系统复杂度的增加，无状态服务的这些优势，会越来越明显。因此，对于一个大型系统而言，我们更推荐无状态化的服务设计。

2. 实践

下面，我们根据不同的服务类型，来分析如何进行状态分离。

常见的状态信息

- **Web 服务：**在 Web 服务中，我们往往需要用户状态信息。一个用户的访问过程，我们称为一个会话（Session），这也是 Web 服务中最为常见的状态信息。
- **本地数据：**在业务运行过程中，会把一些运行状态信息保留到本地内存或者磁盘中。
- **网络状态：**一些服务在配置时，会直接使用 IP 地址访问，这样在服务访问时就依赖相应的网络配置。一旦地址改变，就需要修改对应的配置文件。

状态分离

要把有状态的服务改造成无状态的服务，一般有以下两种做法。

- **请求附带全部状态信息：**这种做法适用于状态信息比较简单的情况（如用户信息，登录状态等）。优点是实现较为简单，不需要额外设施。缺点是会导致请求内容增加，因此在状态信息较多时并不适用。
- **状态分离：**即通过将状态信息分离到外部的独立存储系统中（一般是高速缓存数据库等），来把状态信息从服务中剥离出去。

Web 服务状态分离

在 Web 服务中，两种状态分离模式都可以实现状态分离。

- **使用 Cookie：**把会话信息保存在加密后的 Cookie 之中，每次请求时解析 Cookie 内容来判断是否登录。这种做法的优点是实现简单，不需要额外的系统支持。缺

点是 Cookie 的大小有限制，不能保持较大的状态信息，还会增加每次请求的数据传输量，同时 Cookie 必须要使用可靠的加密协议进行加密，否则会有被人篡改或者伪造的风险。因此这种做法一般用来保持用户登录状态。

- **共享 Session:** 将 Session 信息保存在外部服务（共享内存、缓存服务、数据库等）中，在请求到来时再从外部存储服务中获取状态信息。这种做法没有状态信息大小的限制，也不会增加请求大小。但是需要可靠、高效的外部存储服务来进行支持。一般来说，可以直接使用云计算服务商提供的缓存服务。

服务器本身状态分离

对于依赖本地存储的服务，优先做法是把数据保存在公共的第三方存储服务中，根据内容的不同，可以保存在对象存储服务或者数据库服务中。

如果很难把数据提取到外部存储上，也不建议使用本地盘保存，而是建议通过挂载云硬盘的方式来保持本地状态信息。这样在服务异常时可以直接把云硬盘挂载在其他节点上来实现快速恢复。

对于网络信息，最好的做法是不要通过 IP 地址，而是通过域名来进行访问。这样当节点异常时，可以直接通过修改域名来实现快速的异常恢复。在网易云基础服务中，我们提供了基于域名的服务访问机制，直接使用域名来访问内部服务，减少对网络配置的依赖。

4.2.3 在线水平扩展

在线水平扩展能力是一个分布式系统需要提供的基本能力，也是在架构设计时需要满足的重要功能点。而水平扩展能力也是业务发展的硬性需求，从产品的角度出发，产品的业务流量往往存在着很大波动，具体如下。

- **产品业务量增长:** 在这个信息病毒式传播的时代，一些热点应用的业务量可能会在很短时间内大量增长。
- **周期性业务:** 客服服务、证券服务及春运购票等。这类活动往往都存在着很明显的周期性特征，会按照一定的周期（月、天、小时、分钟）进行波动。波峰和波谷的流量往往会有一个数量级以上的差异。
- **活动推广:** 一次成功的活动推广往往会带来数倍甚至数十倍的流量，需要业务可

以快速扩展，在很短的时间内提供数十倍甚至上百倍的处理能力。

- **秒杀：**秒杀活动是弹性伸缩压力最大的业务，会带来瞬时大量流量。

为了应对这些场景，需要业务在一个很短的时间内提供强大的处理能力。而在业务低谷期，可以相应回收过剩的计算资源，减少消耗，达到系统性能和成本之间的平衡，提高产品的竞争力。

1. 准备工作

产品对水平扩展的需求是一直存在的，但是受制于传统 IT 行业按天甚至按周计算的资源交付能力，弹性伸缩一直是一个美好的愿望。直到云计算这个基础设施完善之后，才使弹性伸缩的实现成为了可能。如果要想实现弹性伸缩，需要以下几点的支持。

- **资源快速交付能力：**业务可以根据需要，动态并快速地申请、释放资源。这也是云计算提供的基础能力，根据云计算平台的不同，一般都会提供从秒级到分钟级的资源交付能力，相对于传统 IT 管理按天计算的交付水平有了巨大的提升，这也是弹性伸缩的基础。在云环境下，绝大部分资源交付、扩容操作都可以在分钟级别完成，从而为弹性伸缩提供了基础支撑。
- **无状态服务设计：**对于有状态服务来说，由于有着各种各样的状态信息，因此会使扩展的难度大大增加。因此，无状态的服务设计，是弹性伸缩服务的前提。
- **业务性能监控：**只有了解业务实际的负载情况，才有弹性伸缩的可能。只有对业务承载能力、运行负载有了全面的了解和实时监控，才能制定出相应的扩展指标。对于云服务厂商来说，基本上都提供了对基础资源，如 CPU、内存、网络等的监控能力。而对应的 PaaS 服务，还提供了应用层数据的详细分析，为更细粒度、更加精确的扩展提供了可能。
- **统一的服务入口：**只有提供了统一的服务入口，才可以在不影响用户的情况下实现后台服务的弹性伸缩。统一服务入口有两种实现机制，一种是在系统层面，通过负载均衡服务提供统一的流量入口，使用负载均衡服务统一进行管理。另一种是通过服务注册和发现机制，在服务层实现适配。对于外部访问，可以使用对外的负载均衡服务，对于内部服务，一般都会提供租户内部的负载均衡。业务方可以根据需要，使用对应的流量入口。

2. 实现要点

前端系统一般都会采用无状态化服务设计，扩展相对简单。在实践中，有多种扩展方案，如通过 DNS 服务水平扩展、使用专有的 apiserver、在 SDK 端分流及接入负载均衡等。其中负载均衡方案使用最广，综合效果也最好，可以满足绝大多数场景下的需要，下面就以负载均衡服务为例，介绍前端系统水平扩展的实现要点。

协议选择

负载均衡服务分为 4 层和 7 层服务，这两种并不是截然分开的，而是有兼容关系。4 层负载均衡可以支持所有 7 层的流量转发，而且性能和效率一般也会更好。而 7 层负载均衡服务的优势在于解析到了应用层数据（HTTP 层），因此可以理解应用层的协议内容，从而做到基于应用层的高级转发和更精细的流量控制。

对于 HTTP 服务，建议直接采用 7 层负载均衡，而其他所有类型的服务，如 WebSocket、MySQL 和 Redis 等，则可以直接使用 TCP 负载均衡。

无状态服务

前端系统可扩展性需要在系统设计层面进行保证，较为通用的做法是无状态化的服务设计。因为无状态，所以在系统扩展时只需要考虑外网设施的支持，而不需要改动服务代码。对于有状态的服务，则尽量服务改造把状态分离出去，将状态拆分到可以扩展的第三方服务中去。

高级流量转发

对于 7 层负载均衡来说，由于解析到了协议层，因此可以基于应用层的内容进行流量转发。除了最常用的粘性会话（Sticky Session）外，最常用的转发规则有基于域名和 URL 的流量转发两种。

- 基于域名的流量转发：外网的 HTTP 服务默认使用 80 端口提供，但经常会有多个不同域名的网站需要使用同样一个出口 IP 的情况。这时候就需要通过应用层解析，根据用户的访问域名把同一个端口的流量分发到不同的后端服务中去。域名流量转发是通过解析请求头的 Host 属性实现的，当浏览器通过域名访问时，会自动设置 Host 头。通过程序访问 HTTP API 接口时，一般的第三方库也会设置这个属性，

但如果自己组装 HTTP 请求，则需要主动设置对应的 Host 头。

- 基于 URL 的流量转发：对一些大型网站，或者基于 REST 风格的 API 接口，单纯通过域名进行分流已经无法满足分流要求。此外，还存在着同一个域名的服务，根据 URL 分流到不同后端集群的情况，这种情况就可以通过请求中的 URL 路径信息，进行进一步分流。一般的 URL 都会支持模糊匹配。

4.2.4 后端系统扩展

后端系统，一般指用户接入端（Web 系统、长连接服务器）和各种中间件之后的后台系统。在这一阶段，最重要的后端系统就是两种，缓存服务和数据库服务。下面，我们分别以 Redis 缓存服务和 MySQL 数据库为例，来介绍后端系统水平扩展的技术和核心技术点。

1. Redis 水平扩展

Redis 去年发布了 3.0 版本，官方支持了 Redis cluster 即集群模式。至此结束了 Redis 没有官方集群的时代，在官方集群方案以前应用最广泛的就属 Twitter 发布的 Twemproxy (<https://github.com/twitter/twemproxy>)，国内的有豌豆荚开发的 Codis (<https://github.com/wandoulabs/codis>)。

下面我们介绍一下 Twemproxy 和 Redis Cluster 两种集群水平扩展。

Twemproxy+Sentinel 方案

Twemproxy，也叫 nutcracker，是 Twitter 开源的一个 Redis 和 Memcache 快速/轻量级代理服务器。

Twemproxy 内部实现多种 hash 算法，自动分片到后端多个 Redis 实例上，Twemproxy 支持失败节点自动删除，它会检测与每个节点的连接是否健康。为了避免单点故障，可以平行部署多个代理节点（一致性 hash 算法保证 key 分片正确），Client 可以自动选择一个。

有了这些特性，再结合负载均衡和 Sentinel 就可以架构出 Redis 集群，如图 4-5 所示。

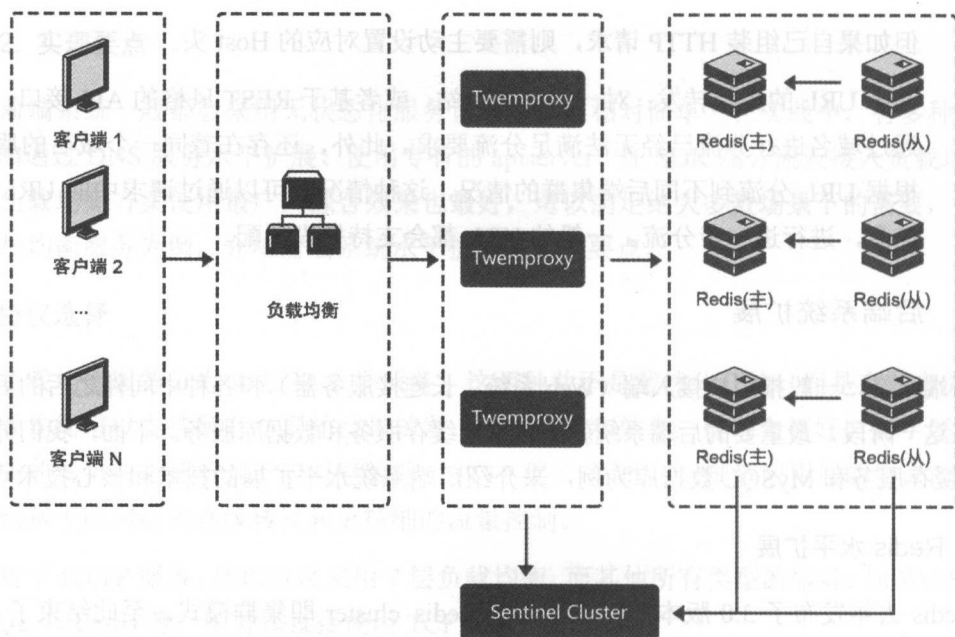


图 4-5 基于 Twemproxy 的 Redis 水平扩展

- 负载均衡：实现 Twemproxy 的负载均衡，提高 proxy 的可用性和可扩展能力，使 Twemproxy 的扩容对应应用透明。
- Twemproxy 集群：多个 Twemproxy 平行部署（配置相同），同时接受客户端的请求并转发请求给后端的 Redis。
- Redis Master-Slave 主从组：Redis Master 存储实际的数据，并处理 Twemproxy 转发的数据读写请求。数据按照 hash 算法分布在多个 Redis 实例上。Redis Slave 复制 master 的数据，作为数据备份。在 Master 失效的时候，由 Sentinel 把 Slave 提升为 Master。
- Sentinel 集群：检测 Master 主从存活状态，当 Redis Master 失效的时候，把 Slave 提升为新 Master。

水平扩展实现就可以将一对主从实例加入 Sentinel 中，并通知 Twemproxy 更新配置加入新节点，将部分 key 通过一致性 hash 算法分布到新节点上。

Twemproxy 方案缺点如下。

- 加入新节点后，部分数据被动迁移，而 Twemproxy 并没有提供相应的数据迁移能力，这样会造成部分数据丢失。
- LB（负载均衡）+ Twemproxy + Redis 3 层架构，链路长，另外加上使用 Sentinel 集群保障高可用，整个集群很复杂，难以管理。

Redis Cluster 方案

Redis Cluster 是 Redis 官方推出的集群解决方案，其设计的重要目标就是方便水平扩展，在 1000 个节点的时候仍能表现良好，并且可线性扩展。

Redis Cluster 和传统的集群方案不一样，在设计的时候，就考虑到了去中心化、去中间件，也就是说，集群中的每个节点都是平等的关系，每个节点都保存各自的数据和整个集群的状态。

数据的分配也没有使用传统的一致性哈希算法，取而代之的是一种叫做哈希槽（hash slot）的方式。Redis Cluster 默认分配了 16384 个 slot，当我们 set 一个 key 时，会用 CRC16 算法来取模得到所属的 slot，然后将这个 key 分到哈希槽区间的节点上，具体算法是 $CRC16(key) \% 16384$ 。举个例子，假设当前集群有 3 个节点，那么：

- 节点 r1 包含 0 到 5500 号哈希槽。
- 节点 r2 包含 5501 到 11000 号哈希槽。
- 节点 r3 包含 11001 到 16384 号哈希槽。

集群拓扑结构如图 4-3 所示，此处不再重复给出。

Redis Cluster 水平扩展很容易操作，新节点加入集群中，通过 redis-trib 管理工具将其其他节点的 slot 迁移部分到新节点上面，迁移过程并不影响客户端使用，如图 4-6 所示。

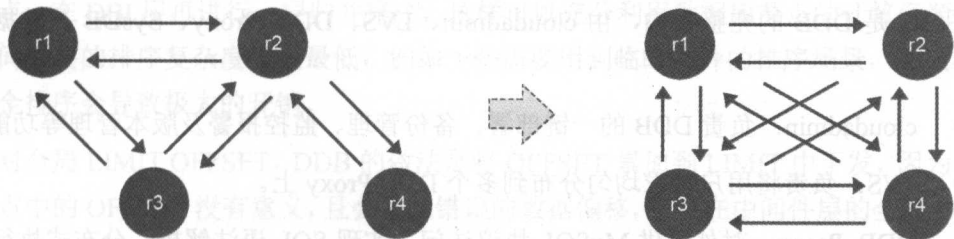


图 4-6 Redis Cluster 水平扩展

为了保证数据的高可用性，Redis Cluster 加入了主从模式，一个主节点对应一个或多个从节点，主节点提供数据存取，从节点则是从主节点实时备份数据，当这个主节点瘫痪后，通过选举算法在从节点中选取新主节点，从而保证集群不会瘫痪。

Redis Cluster 其他具体细节可以参考官方文档，这里不再详细介绍。Redis Cluster 方案缺点如下。

- 客户端实现复杂，管理所有节点连接，节点失效或变化需要将请求转移到新节点。
- 没有中心管理节点，节点故障通过 gossip 协议传递，有一定时延。

2. 数据库水平扩展

单机数据库的性能由于物理硬件的限制会达到瓶颈，随着业务数据量和请求访问量的不断增长，产品方除了需要不断购买成本难以控制的高规格服务器，还要面临不断迭代的在线数据迁移。在这种情况下，无论是海量的结构化数据还是快速成长的业务规模，都迫切需要一种水平扩展的方法将存储成本分摊到成本可控的商用服务器上。同时，也希望通过线性扩容降低全量数据迁移对线上服务带来的影响，分库分表方案便应运而生。

分库分表的原理是将数据按照一定的分区规则 Sharding 到不同的关系型数据库中，应用再通过中间件的方式访问各个 Shard 中的数据。分库分表的中间件，隐藏了数据 Sharding 和路由访问的各项细节，使应用在大多数场景下可以像单机数据库一样，使用分库分表后的分布式数据库。

分布式数据库

网易早在 2006 年就开始了分布式数据库（DDB）的研究工作，经过 10 年的发展和演变，DDB 的产品形态已全面趋于成熟，功能和性能得到了众多产品的充分验证。

图 4-7 是 DDB 的完整架构，由 cloudadmin、LVS、DDB Proxy、SysDB 及数据节点组成。

- cloudadmin：负责 DDB 的一键部署、备份管理、监控报警及版本管理等功能。
- LVS：负责将用户请求均匀分布到多个 DDB Proxy 上。
- DDB Proxy：对外提供 MySQL 协议访问，实现 SQL 语法解析、分布式执行计划生成、下发 SQL 语句到后端数据库节点，汇总合并数据库节点执行结果。

- SysDB: DDB 元数据存储数据库, 也基于 RDS 实现高可用。
- RDS: 底层数据节点, 一个 RDS 存储多个数据分片。

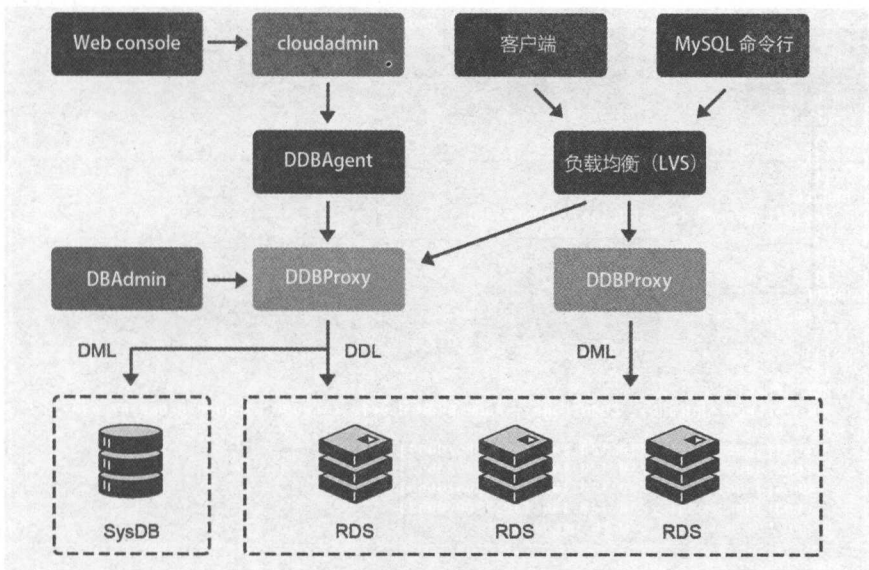


图 4-7 网易 DDB 架构

分布式执行计划

分布式执行计划定义了 SQL 在分库分表环境中各个数据库节点上执行的方法、顺序和合并规则, 是 DDB 实现中最为复杂的一环。如 SQL: `select * from user order by id limit 10 offset 10`。

这个 SQL 要查询 ID 排名在 10~20 之间的 user 信息, 这里涉及全局 ID 排序和全局 LIMIT OFFSET 两个合并操作。对全局 ID 排序, DDB 的做法是将 ID 排序下发给各个数据库节点, 在 DBI 层再进行一层归并排序, 这样可以充分利用数据库节点的计算资源, 同时将中间件层的排序复杂度降到最低, 例如一些需要用到临时文件的排序场景, 如果在中间件做全排序会导致极大的开销。

对全局 LIMIT OFFSET, DDB 的做法是将 OFFSET 累加到 LIMIT 中下发, 因为单个数据节点中的 OFFSET 没有意义, 且会造成错误的数据偏移, 只有在中间件层的全局 OFFSET 才能保证 OFFSET 的准确性。

所以最后下发给各个 DBN 的 SQL 变为: `select * from user order by id limit 20`。又如

SQL: `select avg(age) from UserTest group by name` 可以通过 EXPLAIN 语法得到 SQL 的执行计划, 如图 4-8 所示。

```
mysql> explain select avg(age) from UserTest group by name;
+-----+
| PLAN |
+-----+
| AGGREGATE |
| Do: |
| /\ |
| /|\ |
| || |
| PROJECT |
| Project record to: SUM(age),COUNT(age), |
| /\ |
| /|\ |
| || |
| GROUP |
| Group By: name, |
| /\ |
| /|\ |
| || |
| MERGE-SELECT |
| SQL: SELECT SUM(age), COUNT(age), name FROM UserTest GROUP BY name ORDER BY name ASC |
| Dist Node: |
| dbn1[jdbc:mysql://10.120.146.129:3306/dbn1] |
| dbn2[jdbc:mysql://10.120.146.129:3306/dbn2] |
| dbn4[jdbc:mysql://10.120.146.130:3306/dbn4] |
| dbn3[jdbc:mysql://10.120.146.130:3306/dbn3] |
| Order by: name ASC, with merge sort. |
+-----+
23 rows in set (0.15 sec)
```

图 4-8 分布式执行计划

上述 SQL 包含 GROUP BY 分组和 AVG 聚合两种合并操作, 与全局 ORDER BY 类似, GROUP BY 也可以下发给数据节点、中间件层做一个归并去重, 但是前提要将 GROUP BY 的字段同时作为 ORDER BY 字段下发, 因为归并的前提是排序。对 AVG 聚合, 不能直接下发, 因为得到所有数据节点各自的平均值, 不能求出全局平均值, 需要在 DBI 层把 AVG 转化为 SUM 和 COUNT 再下发, 在结果集合并时再求平均。

DDB 执行计划的代价取决于 DBI 中的排序、过滤和连接, 在大部分场景下, 排序可以将 ORDER BY 下发简化为一次性归并排序, 这种情况下代价较小, 但是对 GROUP BY 和 ORDER BY 同时存在的场景, 需要优先下发 GROUP BY 字段的排序, 以达到归并分组的目的, 这种情况下, 就需要将所有元素做一次全排序, 除非 GROUP BY 和 ORDER BY 字段相同。

DDB 的连接运算有两种实现, 第一种是将连接直接下发, 若连接的两张表数据分布完全相同, 并且在分区字段上连接, 则满足连接直接下发的条件, 因为在不同数据节点的分区字段必然没有相同值, 不会出现跨库连接的问题。第二种是在不满足连接下发条件时,

会在 DBI 内部执行 Nest Loop 算法，驱动表的顺序与 FROM 表排列次序一致，此时若出现 ORDER BY 表次序与表排列次序不一致，则不满足 ORDER BY 下发条件，也需要在 DBI 内做一次全排序。

分库分表的执行计划代价相比单机数据库而言，更加难以掌控，即便是相同的 SQL 模式，在不同的数据分布和分区字段使用方式上，也存在很大的性能差距，DDB 的使用要求开发者和 DBA 对执行计划的原理具有一定认识。

分库分表在分区字段的使用上很有讲究，一般建议应用中 80% 以上的 SQL 查询通过分区字段过滤，使 SQL 可以单库执行。对于那些没有走分区字段的查询，需要在所有数据节点中并行下发，这对线程和 CPU 资源是一种极大的消耗，伴随着数据节点的扩展，这种消耗会越来越剧烈。另外，基于分区字段跨库不重合的原理，在分区字段上的分组、聚合、DISTINCT、连接等操作，都可以直接下发，这样对中间件的代价往往最小。

分布式事务

分布式事务是个历久弥新的话题，分库分表、分布式事务的目的是保障分库数据一致性，而跨库事务会遇到各种不可控制的问题，如个别节点永久性宕机，像单机事务一样的 ACID 是无法奢望的。另外，业界著名的 CAP 理论也告诉我们，对分布式系统，需要将数据一致性和系统可用性、分区容忍性放在天平上一起考虑。

两阶段提交协议（简称 2PC）是实现分布式事务较为经典的方案，适用于中间件这种数据节点无耦合的场景。2PC 的核心原理是通过提交分阶段和记日志的方式，记录下事务提交所处的阶段状态，在组件宕机重启后，可通过日志恢复事务提交的阶段状态，并在这个状态节点重试，如 Coordinator 重启后，通过日志可以确定提交处于 Prepare 还是 PrepareAll 状态，若是前者，说明有节点可能没有 Prepare 成功，或所有节点 Prepare 成功但还没有下发 Commit，状态恢复后给所有节点下发 RollBack；若是 PrepareAll 状态，需要给所有节点下发 Commit，数据库节点需要保证 Commit 幂等。与很多其他一致性协议相同，2PC 保障的是最终一致性。

2PC 整个过程如图 4-9 所示，更详细的过程与问题说明请参考第 5 章的内容，本节仅介绍 DDB 的两阶段分布式事务实现。

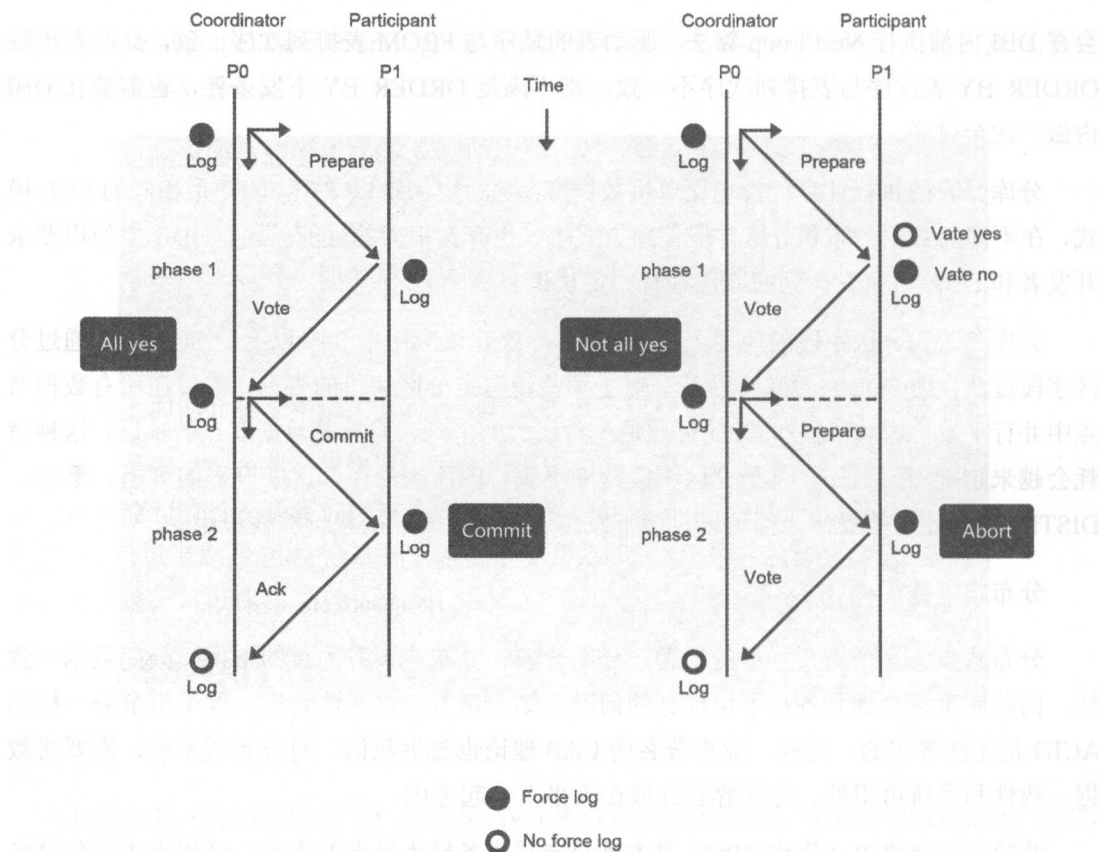


图 4-9 两阶段提交协议

在网易 DDB 中, DBI 和 Proxy 组件都作为 Coordinator 存在, 2PC 实现时, 记录 Prepare 和 PrepareAll 的日志必须 sync, 以保障重启后恢复状态正确, 而 Coordinator 最后的 Commit 日志主要作用是回收之前日志, 可异步执行。

由于 2PC 要求 Coordinator 记日志, 事务吞吐率受到磁盘 I/O 性能的约束, 为此 DDB 实现了 GROUP I/O 优化, 可极大程度提升 2PC 的吞吐率。2PC 本质上说是一种阻塞式协议, 两阶段提交过程需要大量线程资源, 因此 CPU 和磁盘都有额外消耗, 与单机事务相比, 2PC 在响应时间和吞吐率上相差很多, 从 CAP 角度出发, 可以认为 2PC 在一定程度上成全了 C, 牺牲了 A。

另外, 目前 MySQL 最流行的 5.5 和 5.6 版本中, XA 事务日志无法复制到从节点, 这意味着主库一旦宕机, 切换到从库后, XA 的状态会丢失, 可能造成数据不一致, MySQL 5.7

版本在这方面已经有所改善。

虽然 2PC 有诸多不足，我们依然认为它在 DDB 中有实现价值，DDB 作为中间件，其迭代周期要比数据库这种底层服务频繁，若没有 2PC，一次更新或重启就可能造成应用数据不一致。从应用角度看，分布式事务的现实场景常常无法规避，在有能力给出其他解决方案前，2PC 也是一个不错的选择。

对购物转账等电商和金融业务，中间件层的 2PC 最大问题在于业务不可见，一旦出现不可抗力或意想不到的一致性破坏，如数据节点永久性宕机，业务难以根据 2PC 的日志进行补偿。金融场景下，数据一致性是命根，业务需要对数据有百分之百的掌控力，建议使用 TCC 这类分布式事务模型，或基于消息队列的柔性事务框架，请参考第 5 章，这两种方案都在业务层实现，业务开发者具有足够掌控力，可以结合 SOA 框架来架构。原理上说，这两种方案都是大事务拆小事务，小事务变本地事务，最后通过幂等的 Retry 来保障最终一致性。

弹性扩容

分库分表数据库中，在线数据迁移也是核心需求，会用在以下两种场景中。

- 数据节点弹性扩容：随着应用规模不断增长，DDB 现有的分库可能有一天不足以支撑更多数据，要求 DDB 的数据节点具有在线弹性扩容的能力，而新节点加入集群后，按照不同的 Sharding 策略，可能需要将原有一些数据迁入新节点，如 HASH 分区，也有可能不需要在线数据迁移，如一些场景下的 Range 分区。无论如何，具备在线数据迁移是 DDB 支持弹性扩容的前提。
- 数据重分布：开发者在使用 DDB 过程中，有时会陷入困局，比如一些表的分区字段一开始没考虑清楚，在业务初具规模后才明确应该选择其他字段。又如一些表一开始认为数据量很小，只要单节点分布即可，而随着业务变化，需要转变为多节点 Sharding。这两种场景都体现了开发者对 DDB 在线数据迁移功能的潜在需求。

无论是弹性扩容，还是表重分布，都可当作 DDB 以表或库为单位的一次完整在线数据迁移。该过程分为全量迁移和增量迁移两个阶段，全量迁移是将原库或原表中需要迁移的数据 DUMP 出来，并使用工具按照分区策略导入到新库新表中。增量迁移是要将全量迁移过程中产生的增量数据更新按照分区策略应用到新库新表。

全量迁移的方案相对简单，使用 DDB 自带工具按照特定分区策略 DUMP 和 Load 即可。对增量迁移，DDB 实现了一套独立的迁移工具 Hamal 来订阅各个数据节点的增量更新，

Hamal 内部又依赖 DBI 模块将增量更新应用到新库新表，如图 4-10 所示。

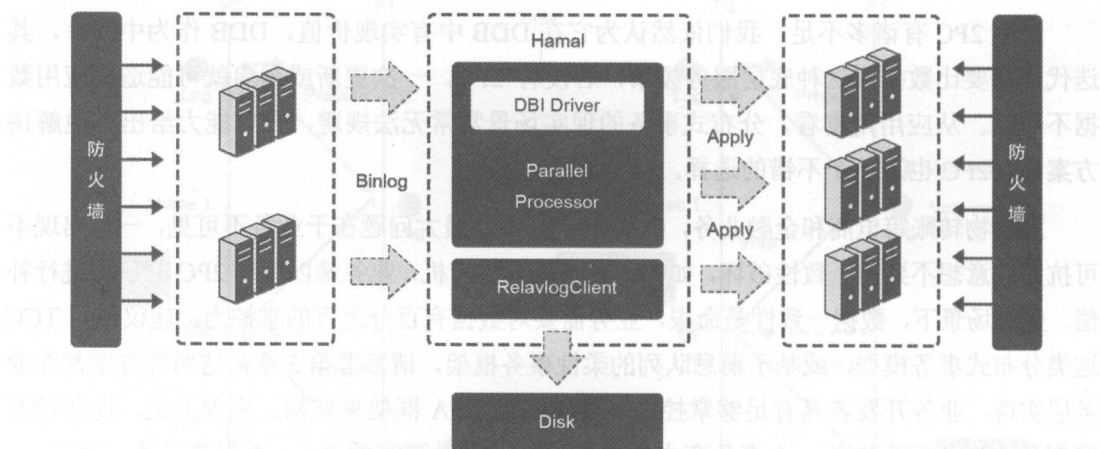


图 4-10 DDB 增量迁移工具 Hamal

Hamal 作为独立服务，与 Proxy 一样由 DDB 统一配置和管理，每个 Hamal 进程负责一个数据节点的增量迁移，启动时模拟 Slave 向原库拉取 Binlog 存储本地，之后实时通过 DBI 模块应用到新库新表，除了基本的迁移功能外，Hamal 具备以下两个特性。

- 并行复制：Hamal 的并行复制组件，通过在增量事件之间建立有向无环图，实时判断哪些事件可以并行执行，Hamal 的并行复制与 MySQL 的并行复制相比快 10 倍以上。
- 断点续传：Hamal 的增量应用具有幂等性，在网络中断或进程重启之后可以断点续传。

全局表

考虑一种场景：City 表记录了国内所有城市信息，应用中有很多业务表需要与 City 做联表查询，如按照城市分组统计一些业务信息。假设 City 的主键和分区键都是 CityId，若连接操作发生在中间件层，代价较高，为了将连接操作下发数据节点，需要让连接的业务表同样按照 CityId 分区，而大多数业务表往往不能满足这个条件。

连接直接下发需要满足两个条件，数据分布相同和分区键上连接，除此之外，其实还有一种解法，可以把 City 表冗余到所有数据节点中，这样各个数据节点本地连接的集合便是所求结果。DDB 将这种类型的表称之为全局表。

全局表的特点是更新极少，通过 2PC 保障各个节点冗余表的一致性。可以通过在建表语句添加相关 Hint 指定全局表类型，在应用使用 DDB 过程中，全局表的概念对应用不可见。

4.2.5 系统通信

1. 系统通信的应用场景

按照应用场景，系统通信可以分为系统间和系统内部两种，从技术角度上看，两种方式没有实质的区别。但在实际场景中，会涉及部署环境、系统兼容等问题，需要根据具体的场景进行分析。

- 系统内通信：随着系统规模的增大，单体架构会越来越复杂，因此必须对原有系统进行拆分和部署。从另外一个角度看，单点的计算能力，可靠性总是有限的，现在的技术方案也倾向于使用集群部署来解决问题。一个系统的多个独立模块之间要协同工作，需要模块之间的通信能力。
- 系统间通信：一个系统不可能完全孤立，越是复杂的系统越是如此。一个复杂系统，除了自己的核心功能外，肯定会依赖各种外部服务来完成各种外围任务。这就涉及与外部系统的通信问题。

系统集成的核心问题是不同系统（模块）之间通信的问题，因此，本节会介绍系统通信所使用的主要技术、优缺点，以及在实践中如何选择。

2. 网络协议选择

最常见的系统间通信方式是直接使用应用层协议消息集成，如图 4-11 所示。根据使用的协议，可以分为基于 HTTP 协议的消息通信、基于 HTTPS 协议的消息通信和基于 TCP 协议的消息通信。

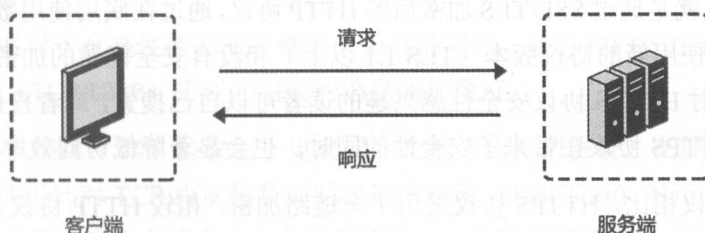


图 4-11 应用层消息集成

基于 HTTP 协议的消息通信

最常见的做法是将 HTTP 协议封装为 RESTful 风格的接口，用来处理系统间的调用。作为系统间通信协议，HTTP 的优点十分明显。

- 语言无关：HTTP 协议本身的语言无关性，可以直接在跨语言系统使用。
- 格式灵活：HTTP 协议对传输内容几乎没有要求，因此用户可以方便地根据需求进行定制和二次封装。
- 良好支持：很多语言都对 HTTP 协议提供了原生支持，而几乎所有语言都有成熟的 HTTP 第三方库。
- 易用性：由于 HTTP 协议的广泛使用，互联网开发人员对 HTTP 协议都有所了解。而 HTTP 协议采用简单的请求/响应 (Req/Res) 模式，学习成本很低，就算不曾接触 HTTP 协议的开发人员也很容易在项目中使用。

正是由于 HTTP 协议有着以上诸多优势，在实际应用中，HTTP 协议也成为现在应用最为广泛的系统间通信方式。而很多第三方框架，更是在 HTTP 协议之上进行了封装，让 HTTP 协议变得更加简单易用。

但是，事情总是相对的。简单易用则表示功能单薄，格式灵活的另一面是数据臃肿，而易使用并不等于易精通，过于广泛的使用也就意味着存在着协议滥用的情况，因此根据具体的业务场景进行分析，选择合适的传输协议十分必要。

HTTP 协议是通过明文传输的，在互联网上毫无安全性可言，因此如果要通过外网进行系统间通信，那么 HTTPS 协议是必要的。

基于 HTTPS 协议的消息通信

HTTPS 协议就是通过 SSL/TLS 加密后的 HTTP 协议，通过在底层使用数据加密，HTTPS 协议可以保证在使用新的协议版本 (TLS 1.1 以上) 和没有安全问题的加密协议的前提下，数据是安全的。对 HTTPS 协议安全性感兴趣的读者可以自己搜索，或者直接查看官方 RFC 文档。但是，HTTPS 协议在带来了安全性的同时，也会显著降低访问效率。

与 HTTP 协议相比，HTTPS 协议采用了全链路加密，相较 HTTP 协议，HTTPS 协议的数据量要大一些，而传输效率更低。这在建立连接时最为明显 (用在协议上的时间会增加数倍)，在传输过程中，由于使用了高效的对称加密协议，因此在通信过程中效率并不会和

HTTP 协议有太大差异。

- HTTPS 协议使用注意事项如下。

保管好私钥，这一点非常重要，HTTPS 安全性的前提是建立在私钥不被泄露的基础上，私钥一旦被泄露，HTTPS 的安全性就毫无意义。更为严重的是，如果没有发现私钥被泄露，第三方攻击者就可以解析网站的任何请求。一旦私钥被泄露，除了更换密钥，没有任何办法。考虑到网站的 HTTPS 证书大多是真金白银买来的，最后再强调一遍——保管好私钥！

- HTTPS 协议配置优化。

在使用 HTTPS 协议时，建议打开 `http keepalive` 功能（`http 1.1` 协议默认开启），这样在使用 HTTPS 协议的时候，除了第一次建立连接的时候稍慢，之后的请求大部分都可以通过缓存绕过握手过程，提高访问效率。

而在使用 HTTPS 服务端时，也建议打开 HTTPS 的会话重用，这样在用户多次访问的情况下，可以大幅减少握手的次数，提高服务端资源占用。

在简单的测试中，使用 Nginx 作为服务端，在开启/关闭 HTTPS 会话重用，并且会话未失效的情况下，服务端性能会有 7~8 倍的性能差异。在实际使用场景中，虽然不会有如此明显的性能提升，但是效果也很明显。

基于 TCP 协议的消息通信

无论是 HTTP 协议还是 HTTPS 协议，都无法解决 HTTP 协议数据臃肿、效率低下的问题。

这一问题的根本原因在于，HTTP 协议（超文本传输协议 HTTP，HyperText Transfer Protocol），就像它的名字一样，在设计之初只为了在网络上传输超文本信息而存在，并不是为了实现服务器间通信，从另一个角度来说，现在 HTTP 协议的广泛使用，很多情况下也是一种滥用。

因此，在那些对于有高性能和实时性要求很高的场景，使用 HTTP 方式通信并不合适，基于传输层协议（TCP/UDP）实现应用层的消息通信更好。

与 HTTP 协议相比，TCP 协议位于更底层，是基于连接的协议。在使用上，大概可以分为以下几种：使用针对 TCP 协议封装好的框架来实现，比如 Thrift, Netty；使用基于 TCP 之上的私有协议实现，这个在很多大小公司的设计上；使用第三方基于 TCP 的标准协议实现，比如针对移动通信优化的 MQTT 协议，针对数据传输的 Google protobuf 协议等。

设计要点

由于使用简单，HTTP 协议使用率很高，但也存在协议被滥用的情况。在一些需要高性能、实时性更高的场合，基于 TCP 的通信协议则更为合适。可以根据业务场景和具体的性能需求来选择。

无论哪种协议，对于消息的可靠性都不会有很好的保证，在实际使用中会出现数据丢失、响应变慢的情况。对于对通信数据可靠性、数据到达有较高要求的应用，可以使用消息队列来解决问题。

4.2.6 消息中间件

消息中间件是分布式系统中重要的组件，如图 4-12 所示，它能够解决应用耦合、异步投递消息、流量削峰等问题，实现高性能、高可用、可伸缩和最终一致性架构，是大型分布式系统不可缺少的中间件。

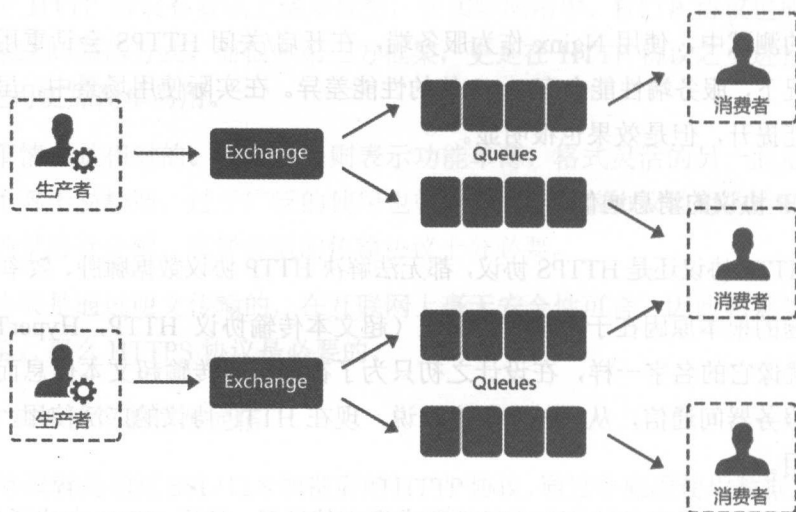


图 4-12 消息中间件

1. 消息模型

队列（Queue）

队列模型下，一条消息只会被一个消费者处理，如图 4-13 所示。在多个消费者的情况

下，生产者投递的消息一般会平均投递到不同的消费者。

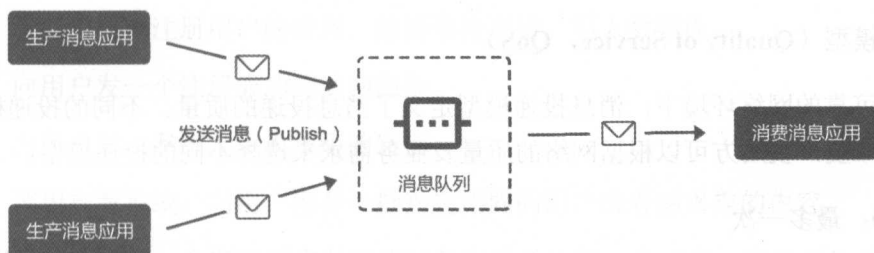


图 4-13 队列

如果使用场景单一，如日志统计，但数据量比较大，需要多个消费者分流，使用队列模型是比较好的选择。

发布订阅 (Pub/Sub)

发布订阅模型下，一条消息会被每一个消费者（或者订阅者）处理，生产者投递的消息会复制到每一个消费者。如图 4-14 所示。

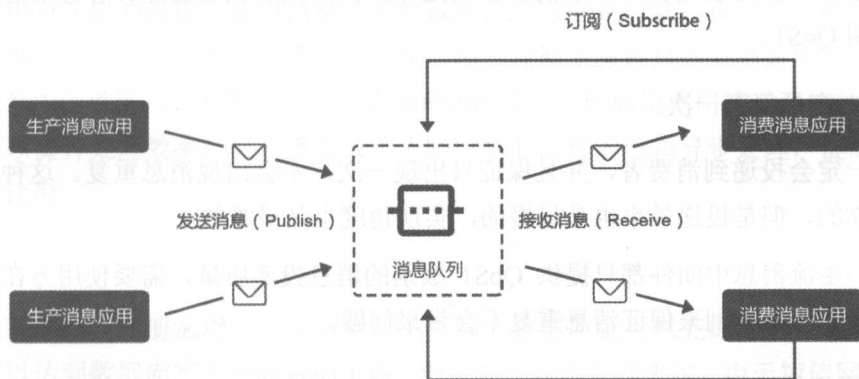


图 4-14 发布订阅

针对一条消息（或者事件）需要做多件事情，如针对用户注册消息，需要发一条确认短信，还需要发一封欢迎邮件，并针对用户资料给用户推荐感兴趣的内容，那么使用发布订阅模型会比较合适。

在上述场景下，针对发送邮件的订阅者，如果一个订阅者处理不过来，需要多个订阅者分流该怎么办？主流的消息中间件针对这种场景，提出了一个消费者群组（Consumer

Group) 的概念。在一个群组内, 遵循队列模型, 而不同群组之间, 遵循发布订阅模型。

投递模型 (Quality of Service, QoS)

在不可靠的网络环境下, 消息投递模型定义了消息投递的质量。不同的投递模型有不同的实现难度, 使用方可以根据网络的质量及业务需求来选择不同的投递模型。

QoS0: 最多一次

只保证消息会尽最大努力投递出去, 不保证消费者一定能收到消息。生产者不会重传消息, 消费者也不会在收到消息时给予确认。一般在你不太关心少量消息是否丢失的场景下可以使用 QoS0。

QoS1: 最少一次

消息至少投递到消费者一次, 消费者需要在收到消息时进行确认, 在没收到消息确认时, 有可能需要重传。因为网络的不可靠性, 确认消息有可能丢失, 导致消费者有可能收到重复消息。在你需要确保每一条消息都被处理, 但不关心或者说能够容忍消息重复的情况下, 使用 QoS1。

QoS2: 有且仅有一次

消息一定会投递到消费者, 并且保证只出现一次, 不会出现消息重复。这种情况下, 消息是可靠的, 但是投递效率也是最慢的, 实现角度也是最难的。

目前, 主流消息中间件都只提供 QoS1 级别的消息投递质量, 需要使用方在处理消息时考虑或者有相应机制来保证消息重复不会带来问题。

2. 使用场景

解耦

消息中间件提供的最主要特性就是解耦。所谓解耦, 就是消息的生产者和消费者可以不需要依赖彼此的存在, 生产者只需要把消息发送到消息中间件就可以, 不需要依赖消费者处理服务一定在线, 等消费者上线以后再处理消息。解耦以后, 使用者只需要关心核心的流程, 依赖其他服务但不那么重要的事情, 就只需要通过消费中间件投递一条通知就行。

举个简单的例子，对于用户注册这个流程来说，一般需要做如下事情。

- 服务端收到注册用户的信息，做简单检查后，写入数据库。
- 向用户发一个注册成功的欢迎邮件。
- 向用户发一条绑定手机号码的短信。
- 调用推荐系统，向用户推荐一批兴趣相同的用户或者感兴趣的内容。

观察这几个流程，会发现用户在注册时其实只关注第一个步骤，即是否注册成功。后续的步骤对于用户注册来说都不是必要的。所以当第一步完成后，就可以向用户返回注册成功，然后把新用户注册成功的信息发送到消息中间件，由相应的处理者（或者消费者）来针对新用户注册成功的信息进行处理，比如邮件发送系统可以针对用户信息向用户发送欢迎邮件，推荐系统根据用户提交的信息，匹配兴趣相同的用户并推荐关注，或者把用户感兴趣的内容推送给用户。

异步化

上述用户注册的过程，使用消息中间件以后，一方面是解耦，事件产生者不需要依赖于事件处理，另一方面产生的效果是异步化。用户注册主流程只需要将新用户注册的事件投递到消息中间件即可，不需要等待具体处理流程完成，比如说不用等欢迎邮件实际发出。异步化可以提升处理效率，将不重要或者不影响核心流程的逻辑异步处理，提高主流程的处理响应速度。

削峰

在之前的用户注册流程中，主流程只需要写入数据库就可以完成，理论上用户注册的吞吐量可以达到数据库写入吞吐量的上限。但是对于短信发送来说，由于短信网关的限制，吞吐量不会很高，和数据库写入吞吐量可能不在一个量级上，但对于用户来说，晚收到绑定手机的短信一般不会有太大问题。消息中间件一般都有堆积消息的能力，在消费者的处理能力跟不上生产者的生产速率时，消息中间件的消息堆积能力就可以提供一定的缓冲，让消费者（比如短信发送系统）平缓处理进入的消息，而不至于压垮整个系统。

3. 实践经验

RabbitMQ 是互联网企业使用比较多的消息中间件，它是基于 AMQP（Advanced

Message Queuing Protocol) 协议的开源实现, 也支持其他协议 (STOMP、MQTT、Websocket 等), 使用广泛。这里我们就以 RabbitMQ 为例简单介绍在实际项目使用过程中常碰到的问题及解决思路。

消息吞吐量

为了支持 QoS1 的消息投递质量, RabbitMQ 提供了生产者确认机制 (confirm)、消费者确认机制 (ack)、持久化机制和高可用机制等。这些机制一方面保证了消息的可靠到达, 但另一方面也会对消息的吞吐量产生较大的影响, 根据我们的测试, 持久化大概会对吞吐量有 40%~50% 的影响, 生产确认机制有 20% 左右的影响, 消费者确认机制有 10% 左右的影响, 而高可用机制大概有 30%~40% 左右的影响。

使用时要根据不同的业务场景选择合适的配置。在测试环境或者对单点故障可以容忍的场景下, 可以选择不使用高可用机制, 但对于不能容忍单点故障的服务来说, 必须使用高可用机制。而对于能够容忍消息丢失或者容忍某些场景下丢失消息的服务来说, 持久化、生产者确认机制、消费者确认机制都可以有选择地使用, 具体要根据不同场景业务的需求, 再结合测试结果选择合适的配置。

资源流控

为了保证 RabbitMQ 服务器正常运行, RabbitMQ 做了一些保护措施, 其中就包括资源 (内存、磁盘) 级别的流控, 当内存占用或者磁盘占用到一定比例时, 就会触发全局流程, 所有生产者将不能发送消息到服务端, 但消费者依然可以处理消息。出现这种情况的原因一般是因为消费速度跟不上生产速度, 要么本身处理速度慢, 要么消费者出现问题时已经停止运行或者已经没在处理消息, 消息大量堆积, 导致内存或者磁盘占用过多。如果因为消费者出现问题导致消息堆积, 就需要尽快排查问题; 如果单纯因为消费速度跟不上, 要么调整消费者所在服务器规格, 要么水平扩展, 添加多个消费者。另外, 在使用 RabbitMQ 的时候, 为了避免因为机器故障导致消费者不可用, 一般情况下, 我们都应该至少部署双副本。不管怎么样, 我们都需要在使用过程中做好监控, 观察生产消费速度是否匹配, 适时调整, 也要做好资源监控, 在流程发生前处理问题, 不影响业务。

网络分区

前面提到高可用机制, 它依赖 RabbitMQ 提供的集群模式, 而 RabbitMQ 的集群模式不

能很好地处理网络分区，所以官方不建议在广域网的环境下使用。但即使在局域网内使用，还是有可能出现网络分区。RabbitMQ 会自己检测网络分区，并在日志或者管理页面提醒使用者已经发生网络分区。发生网络分区时，集群的两个分区都可以正常工作，但两个分区都认为另外一个分区的机器有故障不可用。要从网络分区的状态中恢复，只能选择其中一个分区做为主分区，重启另外一个分区中的所有机器，让它们重新加入集群并同步主分区的状态，但被重启分区中的状态数据会丢失。等重启的分区都加入集群后，要重启整个集群，网络分区的提示才会消失。RabbitMQ 目前提供了网络分区的恢复机制，有以下几个可选项。

- `ignore`: 默认，不做任何处理，如果你的网络非常可靠，比如所有节点连在一个交换机上，可以选择这种模式。
- `pause_minority`: 网络有一点不稳定的情况下使用，重启节点较少的分区。
- `{pause_if_all_down, [nodes], ignore | autoheal}`: 如果分区中的节点不能连接到指定节点，则重启该分区。
- `autoheal`: 网络不稳定的情况下使用，保留客户端连接数比较多的分区，重启其他分区。

支持大量连接

如果将 RabbitMQ 使用在需要支撑大量连接的场景下（比如物联网），就需要做一些调整。一般系统默认都会对打开的文件句柄数有所限制，所以如果要支持大量连接，首先要调整文件句柄数，Linux 下一般可以通过 `ulimit` 来调整。如果对性能有要求，还要进行针对性调优，涉及 RabbitMQ 层面、Erlang 虚拟机（RabbitMQ 使用 Erlang 语言）层面、操作系统层面，以及 TCP 连接层面等，这里不详细展开，具体可以参考官方文档。

4.3 系统优化

系统扩展，着眼点是系统节点的横向扩展，重点是增加系统的整体吞吐量，对于提升用户单次访问没有什么帮助。相反，加入更多横向扩展的中间设施和各种逻辑，往往会减慢而不是加快用户的访问速度。

无论什么产品，最终要为用户服务，因此除了服务端的承载能力之外，用户的访问性

能、产品本身单点稳定也是要考虑的内容。从某种角度上来说，用户访问性能比服务端整体性能更为重要，因为这会切实影响到用户的实际体验。如果说系统扩展是“外功绝学”，重点在于系统的整体表现，系统优化就是“内功心法”，重点提高系统内部的处理能力和稳定性，为用户带来更好的体验。

4.3.1 静态资源分离

静态资源分离是指将静态资源，比如图片、视频、文档、HTML、JavaScript、CSS 与后台应用程序提供的动态查询、页面动态渲染等进行隔离。之所以进行隔离是静态资源和动态资源具有各自不同的特点，需要分别进行优化。

静态资源有以下特点。

- 几乎从不修改、可以长期缓存：图片、视频发布之后就基本不会修改或者删除。
- 带宽要求高、流量较大：互联网上绝大部分流量是由静态资源的访问引起的，典型的有视频网站的流量、电商网站海量的商品图片等。

动态资源有以下特点。

- 可变数据，不能缓存，或只能短期缓存：动态资源请求譬如查询当前用户购物车的商品列表、查询商品的库存信息、微博上面的评论列表。一般都涉及结构化数据或者半结构化数据的访问，为了减小数据库的访问压力，会使用 Redis、Memcached 这类缓存组件进行内存缓存，但是这些数据在自身的生命周期中是经常变化的，如果发生变化，我们需要及时维护这些缓存信息的新鲜度。
- 带宽要求低、QPS 要求高：此类数据的访问带宽需求相比于图片视频完全不在一个量级，一个请求基本都在 1K 以内，但是由于不能缓存，对底层缓存和存储系统的 QPS 要求相对而言就会比较高。

基于动态资源和静态资源各自不同的特点，动静资源的最佳实践就是进行隔离。静态资源基本上都可以托管在对象存储系统上，譬如网易云对象存储服务 NOS、AWS S3 等。

隔离的优势如下。

- 提升性能。
- 更好的扩展性。

- 静态资源服务和业务服务可以单独部署。

静态资源分离实践，包括使用独立域名、避免域名绑定、CDN 服务等。

1. 使用独立域名

在一些大型网站中，我们发现静态资源都放在不同的域名下面，这样做有很多原因。

浏览器并发限制

浏览器对单个域名下的访问有并发限制，现在大部分浏览器都是使用 HTTP 1.1 的模式，HTTP 1.1 的典型交互方式为 ping-pong，在广域网（公网）场景下并不能很好地利用网络的带宽。如图 4-15 所示。

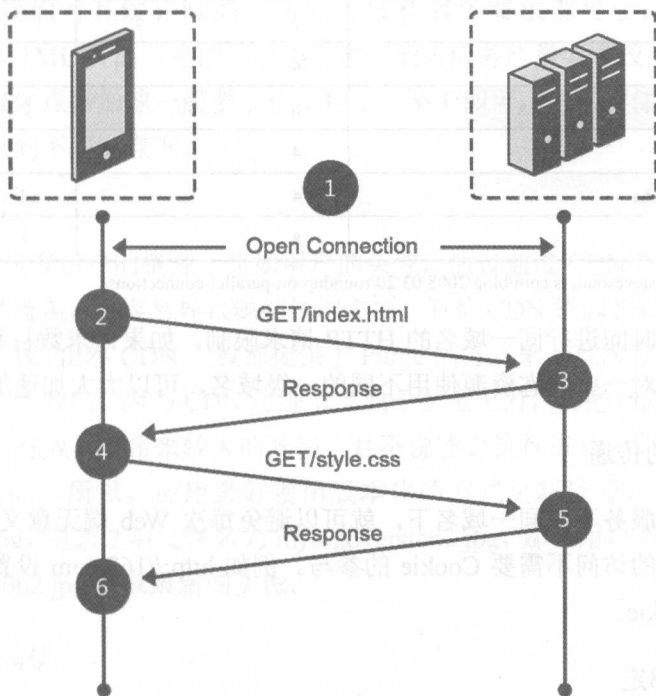


图 4-15 HTTP 1.1 模式的并发限制

参考

为了提升网站的加载速度，浏览器通常都会针对单个域名打开并发的几个 TCP 连接。

表 4-3 为不同浏览器版本对并发的支持情况。

表 4-3 不同浏览器版本对并发的支持

| Browser | HTTP/1.1 | HTTP/1.0 |
|-----------------------|----------|----------|
| IE 6,7 | 2 | 4 |
| IE 8 | 6 | 6 |
| Firefox 2 | 2 | 8 |
| Firefox 3 | 6 | 6 |
| Safari 3,4 | 4 | 4 |
| Chrome 1,2 | 6 | ? |
| Chrome 3 | 4 | 4 |
| Chrome 4+ | 6 | ? |
| iPhone 2 | 4 | ? |
| iPhone 3 | 6 | ? |
| iPhone 4 | 4 | ? |
| Opera 9.63,10.00alpha | 4 | 4 |
| Opera 10.51+ | 8 | ? |

数据来源：<http://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections/>

并发即为同一时间进行同一域名的 HTTP 请求限制。如果请求数目超出限制，则会阻塞。因此，网站中对一些静态资源使用不同的一级域名，可以大大加速加载速度。

避免 Cookie 的传递

静态资源与主服务不在同一域名下，就可以避免每次 Web 端无意义的 Cookie 传输，因为一般静态资源的访问不需要 Cookie 的参与。例如 <http://163.com> 设置 Cookie，所有子域名都会带上 Cookie。

2. 避免域名绑定

如使用第三方厂商提供的静态资源托管服务，最好不要用厂商提供的域名，否则很容易被限定在一个托管服务商上。应该用自己的域名绑定到相应的静态资源托管服务商，因为谁也不能保证谁比谁活得更长久，假如托管方由于各种原因停止服务，那么我们至少可以直接把数据迁移到其他可用的托管厂商，直接将域名切换到新厂商，继续提供服务。

3. 使用 CDN 缓存

还有很关键的一点是静态资源可以通过 CDN（Content Delivery NetWork）加速。源站内容同步到全国各边缘节点，配合精准的调度系统，将用户的请求分配至最适合的节点，比如北京的用户直接在北京城域网就近访问，用户可以以最快的速度取得所需的内容，解决网络带宽小、用户访问量大、网点分布不均等问题，提高用户访问的响应速度。各大云计算厂商一般都会提供 CDN 服务，各大公有云厂商大都提供了自助化平台帮助用户更方便地使用 CDN 进行静态资源加速。使用 CDN 的资源还需要注意以下两点。

● 资源划分

静态资源类型很多，比如 JavaScript 脚本、HTML 静态页面、图片、视频等，对于不同类型的文件，我们对其缓存时间、访问方式和服务要求都可能是不一样的。譬如 JavaScript 脚本、HTML 页面、小图片这类资源一般对服务质量要求较高，它们展现快速，文件较小。视频媒体点播加速一般要求传输稳定、不卡即可，它们存储量较大。所以这些静态资源最好划分到不同站点下。

● 版本化管理

由于 CDN 是缓存用户的资源，比如用户的头像，假设通过 CDN 进行加速，如果用户直接使用同名文件覆盖，很容易导致源站资源更新，但是 CDN 资源长期更新不了，用户看不到最新上传的头像。虽然 CDN 一般都提供了 Purge 的方式来清理缓存的资源，但是 Purge 操作相对来说成本比较高，因为 CDN 边缘节点成千甚至上万，要把 Purge 请求分发到所有的节点并保证执行有效，会带来较大的开销，且不说能否执行成功，即便执行成功也需要 10 分钟左右的时间。所以，应用最好使用版本化的方式更新资源，比如刚开始头像为 mylogo-version1.jpg，更新之后文件名为 mylogoversion2.jpg，那么用户下次访问的时候可以访问 mylogo-version2.jpg 获取最新的头像。

4.3.2 数据库调优

随着运营推广的开始，业务进入快速增长期，数据库作为后端系统唯一或者主要持久化组件，无论是存储的数据量还是事务请求次数都呈现大幅增长，数据库的事务处理能力逐渐成为整个系统性能瓶颈。增加物理资源虽然可以起到一定程度的缓解作用，但是毕竟是一种治标不治本的方法。分布式数据库虽然听起来高端，但是其系统改造成本及学习运

维成本又让一般的中小型团队望而却步。SQL 优化，根据用户访问的 SQL 语句，对数据库的表结构，尤其是索引进行优化，能够有效加速 SQL 的执行效率，对于开发者来说是最简单有效的解决方案。接下来，我们就来聊聊 SQL 优化的一般步骤。

1. 收集 SQL 语句

SQL 优化的起点是用户访问数据库的 SQL 语句，尤其是问题 SQL 语句，在数据库中，主要指访问时间比较长的 SQL 语句。MySQL 数据库提供了慢 SQL 日志功能，帮助开发者获取执行时间超过一定阈值的 SQL 语句列表，通过参数 `slow_query_log` 开启。MySQL 支持两种慢 SQL 日志保存格式：Table 和 File，通过参数 `log_output` 进行配置，使用 table 的优势在于可以直接使用 SQL 进行查询分析。开启慢 SQL 日志后，系统会将执行时间超过 `long_query_time` 的 SQL 语句记入慢 SQL 日志中，默认时间为 10s。

除了超时 SQL 语句，开发者还可以将没有使用索引的 SQL 语句列为问题 SQL，记入慢 SQL 日志，通过 `long_queries_not_using_indexes` 开启。为了控制慢 SQL 日志写入频率，`log_throttle_queries_not_using_indexes` 规定了每分钟因为没有走索引而记入慢 SQL 日志的查询数。MySQL 还提供了参数 `log_slow_admin_statements` 控制管理类型的 SQL 语句是否记入慢 SQL 日志，例如 ALTER TABLE、ANALYZE TABLE 等。`min_examined_row_limit` 规定了所有记入慢 SQL 日志的 SQL 语句最小的行扫描记录数，默认值为 0，只有扫描记录数大于该值的 SQL 语句才会被记入慢 SQL 日志，推荐开发者将其设置为 100，如果 SQL 语句只扫描一行记录，说明该 SQL 语句的执行效率非常高。

图 4-16 展示了使用 SQL 查看慢日志的结果，`sql_text` 字段展示了具体的慢 SQL 语句。

```
mysql> select * from mysql.slow_log \G;
***** 1. Row *****
start_time: 2016-11-07 15:52:01.067393
user_host: guoyi[guoyi] @ [114.113.202.75]
query_time: 00:00:00.103886
lock_time: 00:00:00.000037
rows_sent: 100
rows_examined: 300
logical_reads: 0
physical_reads: 0
db: blog
last_insert_id: 0
insert_id: 0
server_id: 8361
sql_text: SELECT DISTINCT c FROM sbtest1 WHERE id BETWEEN 4964 AND 4964+99 ORDER BY c
thread_id: 885823
1 row in set (0.00 sec)
```

图 4-16 使用 SQL 查看慢日志

慢 SQL 作为衡量数据库访问速度的一个重要指标,可以通过执行 `show global status like 'Slow_queries'` 来监控慢 SQL 数量的变化。如果在某个时间段范围内,数据库的慢 SQL 数量出现急剧增长,开发者就需要关注该时间段内的慢 SQL 语句,进行 SQL 语句排查。

除了慢 SQL 语句,系统中执行次数最多,扫描记录数最多、执行时间最长的 SQL 语句,都是 SQL 优化的对象。在 MySQL 5.7 版本的 `sys` 库的 `statement_analysis` 视图中,收集了所有来自用户的 SQL 语句,如图 4-17 所示。

```
mysql> select * from statement_analysis order by exec_count desc limit 10 \G;
***** 1. row *****
      query: SELECT `c` FROM `sbtest1` WHERE `id` = ?
         db: blog
      full_scan:
      exec_count: 55116057
      err_count: 0
      warn_count: 0
      total_latency: 4.44 h
      max_latency: 273.74 ms
      avg_latency: 290.22 us
      lock_latency: 54.16 m
      rows_sent: 55117923
      rows_sent_avg: 1
      rows_examined: 55118056
      rows_examined_avg: 1
      rows_affected: 0
      rows_affected_avg: 0
      tmp_tables: 0
      tmp_disk_tables: 0
      rows_sorted: 0
      sort_merge_passes: 0
      digest: 88d0c983330b72f801cbb8c63d8a54f5
      first_seen: 2016-11-04 11:20:13
      last_seen: 2016-11-08 10:59:58
```

图 4-17 MySQL 5.7 版本的 `sys` 库的 `statement_analysis` 视图

`query` 字段中的 SQL 语句已经进行了格式化处理,将格式相同仅参数不同的 SQL 语句归为一类。开发者按照 `exec_count`、`latency`、`rows_examined` 3 个字段分别排序,即可获取执行次数最多、执行时间最长、扫描记录数最多的 SQL 语句列表。

2. 跟踪执行过程

明确了待优化的 SQL 语句之后,接下来,为了跟踪这些 SQL 语句执行过程,我们需要采集一些关键指标。

- 索引：如图 4-18 所示，通过执行 explain SQL 语句，获取 SQL 的执行计划，key 字段标识了该 SQL 语句是否通过索引扫描记录，如果该字段为 NULL，且扫描记录数较多，则可以通过创建索引来优化执行效率。

```
mysql> explain select distinct c from sbtest1 where id between 4964 and 4964+99 order by c \G;
***** 1. tqw *****
      id: 1
    select_type: SIMPLE
        table: sbtest1
    partitions: NULL
         type: range
possible_keys: PRIMARY
          key: PRIMARY
        key_len: 4
         ref: NULL
        rows: 100
   filtered: 100.00
    Extra: Using where; Using temporary; Using filesort
1 row in set, 1 warning (0.00 sec)
```

图 4-18 通过执行 explain SQL 语句获取 SQL 的执行计划

- 扫描记录数：慢日志的 rows_examined 字段，标识了该 SQL 语句扫描的记录数，扫描记录数越多，表示该 SQL 语句执行花销越大，执行效率越低。
- 持锁时间：慢日志的 lock_time 字段标识了该 SQL 语句因为锁等待浪费的时间，如果该值较大，说明 SQL 语句存在较多的锁冲突和等待。
- 返回记录数：慢日志的 rows_sent 字段标识该条 SQL 语句返回的记录条数，返回记录越多，对系统资源的消耗也越多。

除了上述几个关键指标外，在网易维护的 MySQL 分支版本 InnoDB 中，还增加了针对某条 SQL 语句的 I/O 开销统计，在 slow log 中增加了 logical reads 和 physical reads 两个列，分别表示该 SQL 语句读数据库缓存和产生实际硬盘 I/O 的次数。

3. 分析优化方案

掌握了这些关键运行指标，接下来就开始分析优化。

- 没有覆盖索引：对于没有被索引覆盖的 SQL 语句的过滤条件涉及的字段，在区分度较大的字段上创建索引，如果涉及多个字段，尽量创建联合索引。需要注意的是，SQL 语句在一些情况下是无法使用索引的，例如使用 !=、<、> 判断等，此时应修改 SQL 语句逻辑。

- SQL 语句被索引覆盖，但是扫描记录数非常多，返回记录数不多：此时要考虑索引是否高效，衡量索引效率的一个判断标准是索引的区分度，通过 MySQL 库下的 `innodb_index_stats` 表的 `stat_value` 字段，我们可以知道该表在该索引涉及列上取值不同的记录数，然后与 `n_rows` 表中记录的该表总记录数相除，即可得到该表的区分度，结果越接近 1，表示区分度越高，结果低于 0.1，则说明区分度较差，开发者应该重新评估 SQL 语句涉及的字段，选择区分度高的多个字段创建索引。
- SQL 语句被索引覆盖，扫描记录数非常多，返回记录数也非常多：此时除了索引效率的问题，也有可能是因为 SQL 语句本身过滤条件不强，导致返回的记录数过多引起的，此时，开发者应该从业务层修改 SQL 语句，增加 SQL 过滤条件。

除了上述指标的特征外，由于索引的创建在插入、更新、删除数据时会带来索引维护的开销，所以我们必须尽可能精简索引的数量，可以通过如下方式实现。

- 去除冗余索引：通过 MySQL `sys` 库下的 `schema_redundant_indexes` 视图，查看当前实例有哪些冗余索引。
- 合并多个索引：对于已经创建的索引，如果多个索引涉及字段顺序一致，且索引的第一个字段相同，则可以组成一个联合索引；例如索引 `(a, c)` 和索引 `(a, d)`，可以组成联合索引 `(a, c, d)`。值得注意的是，合并的两个索引的第一个字段必须相同，因为联合索引的第一个字段必须要出现在查询条件中，否则查询不能使用该索引。
- 去除无效索引：通过 MySQL `sys` 库下的 `schema_unused_indexes` 视图，可以查看当前实例哪些索引从未被使用。

网易云 SQL 优化专家系统

为了方便开发者简单有效地完成 SQL 优化过程，网易云基础服务推出了 SQL 优化专家系统，集成网易 10 年的数据库优化经验进行标准化输出，推出了慢 SQL、SQL 排行榜和热表 3 个功能，如图 4-19 所示。

点击“慢 SQL”，开发者可以查看数据库实例每秒慢 SQL 数量变化，平台自动标识超过警戒线的时间范围，同时还提供了对应时间范围内 CPU 和 I/O 负载变化，如图 4-20 所示。



你的数据库实例运行状况良好，请继续保持！

共检查了 21 项，未发现问题 [查看详情](#)

SQL Profiler

实用工具



热表



SQL 排行榜



慢 SQL



在线修改表结构



WebSQL



表结构对比

图 4-19 网易云 SQL 优化专家系统

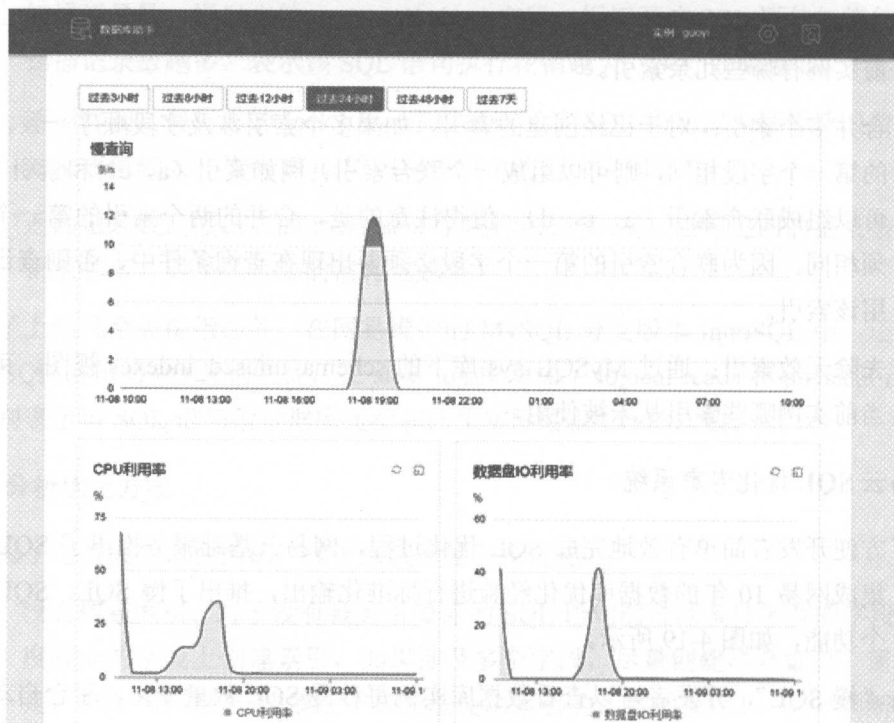


图 4-20 慢 SQL 监控信息

点击某个时间范围，就获取了对应时间范围内的慢 SQL 列表，平台对所有慢 SQL 语句进行了格式化处理，如图 4-21 所示。

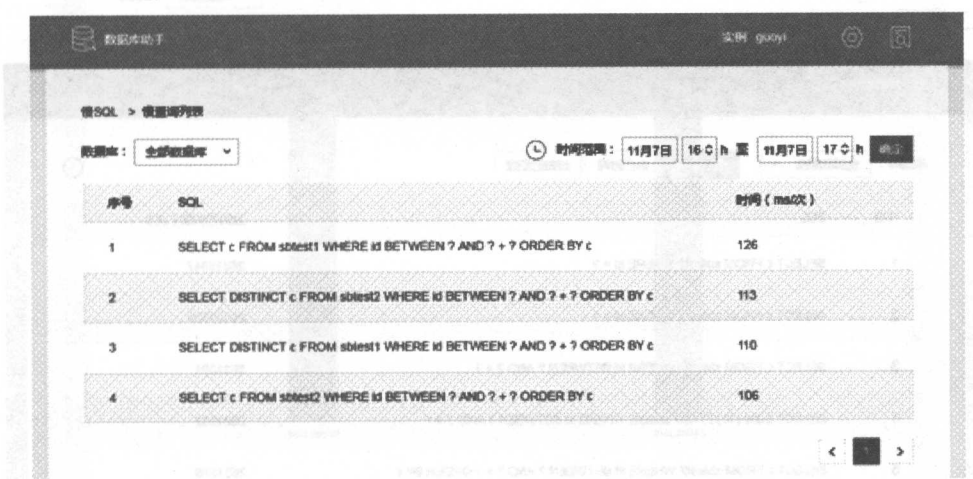


图 4-21 慢 SQL 列表

点击某类 SQL 语句，平台会为开发者呈现该 SQL 语句执行过程中涉及的所有关键指标，同时综合这些关键指标和系统提前预置的优化经验判断逻辑，给出开发者实用的 SQL 优化建议，如图 4-22 所示。

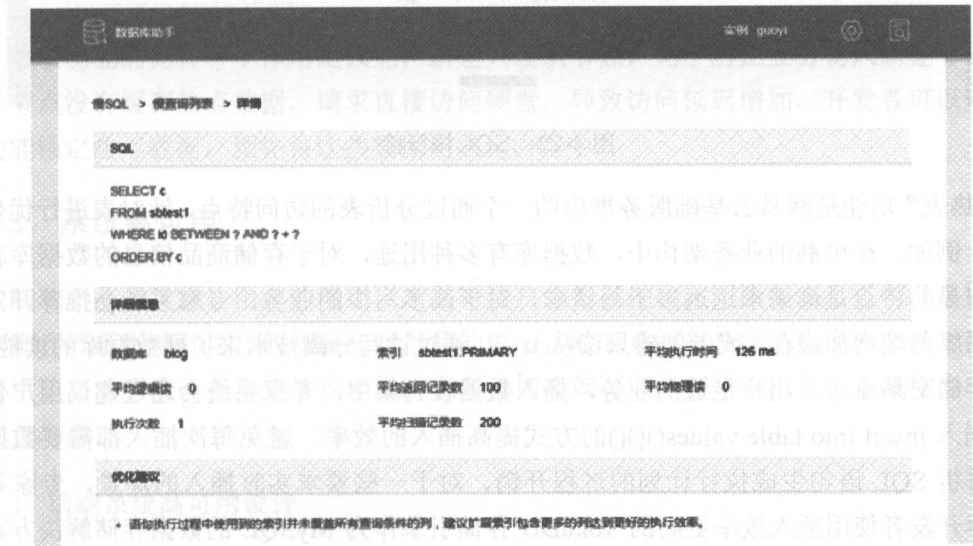


图 4-22 慢 SQL 详细信息

“SQL 排行榜”功能从执行次数、执行时间、扫描记录数 3 个维度对 SQL 语句进行排序，开发者可以轻松获取执行次数最多、执行时间最长、扫描记录数最多的 SQL 语句，如图 4-23 所示。



| 排名 | SQL | 总执行次数 (次) |
|----|---|-----------|
| 1 | SELECT c FROM sbtest2 WHERE id = ? | 26315947 |
| 2 | SELECT c FROM sbtest1 WHERE id = ? | 26301589 |
| 3 | SELECT c FROM sbtest2 WHERE id BETWEEN ? AND ? + ? | 2631591 |
| 4 | SELECT SUM (K) FROM sbtest2 WHERE id BETWEEN ? AND ? + ? | 2631582 |
| 5 | SELECT c FROM sbtest2 WHERE id BETWEEN ? AND ? + ? ORDER BY c | 2631578 |
| 6 | SELECT DISTINCTROW c FROM sbtest2 WHERE id BETWEEN ? AND ? + ? ORDER BY c | 2631569 |
| 7 | UPDATE sbtest2 SET k = k + ? WHERE id = ? | 2631162 |
| 8 | UPDATE sbtest2 SET c = ? WHERE id = ? | 2630804 |
| 9 | INSERT INTO sbtest2 (id , k , c , pad) VALUES (, , ,) | 2630344 |
| 10 | DELETE FROM sbtest2 WHERE id = ? | 2630344 |

图 4-23 SQL 排行榜

“热表”功能是网易云基础服务推出的一个通过分析表的访问特点，针对表进行优化的功能。例如，在电商的业务架构中，数据库有多种用途，对于存储商品信息的数据库表，一个明显的特点是读请求远远多于写请求，对于读多写少的业务，专家系统会推荐开发者在数据库前端增加缓存，或者创建只读从节点，通过读写分离技术来扩展数据库的读能力。对于存储交易流水、用户足迹的业务，插入数据较为集中，专家系统会通过建议用户使用批量插入 insert into table values()()()的方式提高插入的效率，避免每次插入都需要数据库反复解析 SQL 语句生成执行计划的过程开销。对于一些要求实时插入的场景，专家系统会推荐开发者使用插入效率更高的 TokuDB 存储引擎作为 MySQL 的数据存储解决方案，如图 4-24 所示。



图 4-24 热表分析

热表功能的另外一个作用是预热, 新建只读从节点, 为了防止业务切入新建节点后, 由于节点没有缓存热点数据, 请求直接访问硬盘, 导致访问延迟增加, 开发者可以通过热表功能锁定热点数据, 预先将这些数据扫描到内存中。

4.3.3 系统高可用

随着产品的发展、用户的增加, 产品对用户的重要性也在不断提高。任何业务的异常中断都会严重影响到用户的体验和对产品的信心, 随着系统复杂度的提高, 产品对高可用也有了更高的要求, 不再是节点宕机时有备机切换就可以, 而是要把整个系统作为一个整体通盘考虑。

1. 前端系统高可用设计

接入负载均衡, 实现多点部署只是完成了 Web 系统高可用的第一步, 如果设计不合理,

或者实际部署中有问题，在系统发生故障时，单是多点部署无法做到良好的高可用的。要实现系统高可用，还要注意以下几点。

- 余量保证：在这一阶段，系统性能也会成为考虑因素之一，我们会通过多点部署来平衡负载，提高系统容量。但是，在做容量规划的时候，宕机余量是必须考虑的问题。节点总数为 n ，我们容许的最大宕机节点数为 r ，那么系统的安全负载为 $(n-r)/n*100\%$ 。
- 节点数目：2 个副本是一个高可用服务的最低要求。但对于核心的无状态服务来说，是要达到一个可靠的高可用率，4 个节点是比较理想的数目。当只有 2 个节点时，1 个节点宕机就意味着系统失去了一半的处理能力，这在平时可能并不会导致严重问题。但是节点宕机和流量切换都是在系统异常时产生的，这种时候系统整体负载往往也处于异常状态，而 50% 的处理能力损失很有可能成为压垮骆驼的最后一根稻草。当有 4 个节点时，损失 1 个节点只会损失 1/4 的处理能力，而 2 个节点同时宕机的概率要远远小于单节点宕机的概率。因此，对于系统的核心服务，如果要通过多节点实现高可用，4 个节点是比较理想的数目。同理，对于需要内部选举的高可用服务，一般节点数目为单数的情况下，3 个节点是最小要求，但是考虑到系统容错，5 个节点是比较理想的数目。
- 平滑升级：虽然在大多数 SLA 中，挂出公告的维护都不算在系统不可用时间中。但是对于用户体验而言，系统维护确实是实打实的不可用时间。在接入负载均衡的服务下，一般可以通过滚动升级的方式来实现完全平滑的升级，基本思路是在负载均衡端移除需要升级的节点，独立升级并测试，完成后接入负载均衡，再升级下一个节点，直到全部节点升级完毕。这种升级方案可以做到不停机发布。但是也有一个前提，就是新版老版接口必须兼容，否则就会造成数据不一致。
- 动态调度高可用：这种高可用方案依赖资源的快速分配和获取能力，因此只有在云计算场景下才能实现。通过接入云计算服务商提供的 API 接口，来获取系统的运行状态，在系统异常时通过接口来动态地补充资源，恢复异常服务，由此来实现系统高可用。实际上，很多云计算厂商都对这一功能进行了特定峰值，成为了面向固定场景的高可用服务，比如弹性伸缩服务，Kubernetes 中的 RC 服务等。借助于云计算厂商提供的 OpenAPI 接口，我们可以简单实现对应的功能。

2. 数据库高可用

数据库作为后端系统唯一或者主要的持久化组件，数据库实施高可用部署可以有效避免因系统或者硬件故障造成的服务不可用。目前，针对 MySQL 数据库，常用的高可用技术实现包括以下几点。

- 共享存储：基于专用设备实现多个服务器之间的数据文件共享，一旦主服务器宕机，可以将共享存储设备连接到备用服务器上，在备用服务器使用相同的数据文件进行恢复，优点由于是基于同一份数据文件进行恢复，所以可以确保故障恢复前后数据完全一致，缺点是设备价格昂贵，实施相对复杂，一般中小型企业难以承受，业界成熟的方案包括 NAS、SAN。
- DRBD：基于 Linux 内核实现的块设备级别的复制技术，能够实现两个服务器的块设备之间的数据同步，基于同步复制协议模式，可以实现数据严格同步，确保数据不丢，但是对性能影响较大；同样，在主服务器宕机的场景下，备服务器可以基于备服务器上的块设备进行恢复，优点是相比共享存储，不需要专用设备，实施成本较低，缺点是性能较差。
- 复制：复制是 MySQL 数据库最重要的一个特性，在物理资源完全隔离的两个数据库实例之间，通过二进制日志，主实例上的事务更新同步到备实例上，一旦主实例宕机，就将业务切换到备实例上，实现快速故障转移，该实现方案的优点是实施简单，能够做到秒级故障切换，缺点是默认 MySQL 复制是异步模式，主机上的事务提交无需等待从机的响应，所以存在丢数据的可能，另外由于从机二进制日志默认是单线程回放，所以主从之间容易出现复制延迟，从而造成切换时间较长。
- MMM：一组脚本用于实现 MySQL 数据库的监控和基于 vip 的故障切换，数据同步依靠 MySQL 的复制，一旦主机宕机，脚本会自动探测，并将主机上的 vip 漂移到从机上，MMM 的优点是实现了 MySQL 数据库自动宕机检测和切换，缺点是没有考虑切换前后数据一致性。
- MHA：MHA 相比 MMM，同样提供了 MySQL 数据库的自动宕机检测和恢复功能，但是增加了数据一致性补齐的过程，可以确保切换前后数据完全一致；MHA 引入了 manager 节点，在主实例宕机后，通过 SSH 登录到主实例所在服务器，复制二

进制文件，通过和从机上接收到的二进制日志进行对比，将还没有来得及传输的部分应用到从机，从而确保主从数据完全一致，但是如果在主服务器也无法登录，同样存在丢数据的可能。

- **PXC/Galera Cluster:** 实现了 MySQL 多点写入和集群节点数据严格同步，在 cluster 中，每个节点都可以接收事务写请求，首先在本节点执行事务更新，待事务提交时，将事务的 GTID 广播给同一个 cluster 的所有成员，确保成员验证通过可以执行以后，再在主节点提交，如果有一个成员节点不通过，事务就需要全局回滚。优点是多点写入，可以确保数据完全一致，缺点是性能较差，多个节点并发写入，锁冲突严重，事务回滚较为频繁。
- **NDB Cluster:** MySQL 官方出品的一个内存存储引擎，可以实现数据多副本同步，包括 SQL 节点、管理节点和 NDB 数据节点，管理节点负责管理集群内所有节点的状态监控和协调，SQL 节点提供 SQL 解析、连接管理、执行计划生成，NDB 数据节点负责数据的存储，每个 NDB 节点保存的数据通过参数配置可以在另外一个节点存在副本，从而确保数据多副本存储。优点是官方出品，稳定性有保障，能够确保数据完全一致，缺点是操作复杂，同时与现有的 InnoDB 存储引擎不兼容。

开发者在选择数据库的高可用方案时，应该从实施成本、数据可靠性、恢复时间、性能影响、成熟度和通用性角度进行比较，共享存储设备成本高昂，实施代价较高，DRBD 对性能影响严重，MMM 不保障切换数据一致性，MHA 虽然能够确保大多数场景下数据一致，但是在特定服务器宕机场景下，也存在丢数据的可能，PXC/Galera Cluster 方案性能较差，同时存在并发更新锁冲突严重问题，NDB 对 InnoDB 不兼容，通用性较差，所以适合互联网企业或者拥抱互联网+的传统企业的 MySQL 高可用方案是基于复制的主从热备方案，目前国内的主要公有云服务商也均采用该技术方案。

3. 数据一致性

MySQL 复制默认是异步模式，即所有主节点的事务提交无需等待从节点确认，如图 4-25 所示。异步复制模式下，如果主节点宕机，而主节点上已经提交的二进制日志还没来得及传递给从节点，就会导致已经提交的事务更新丢失。

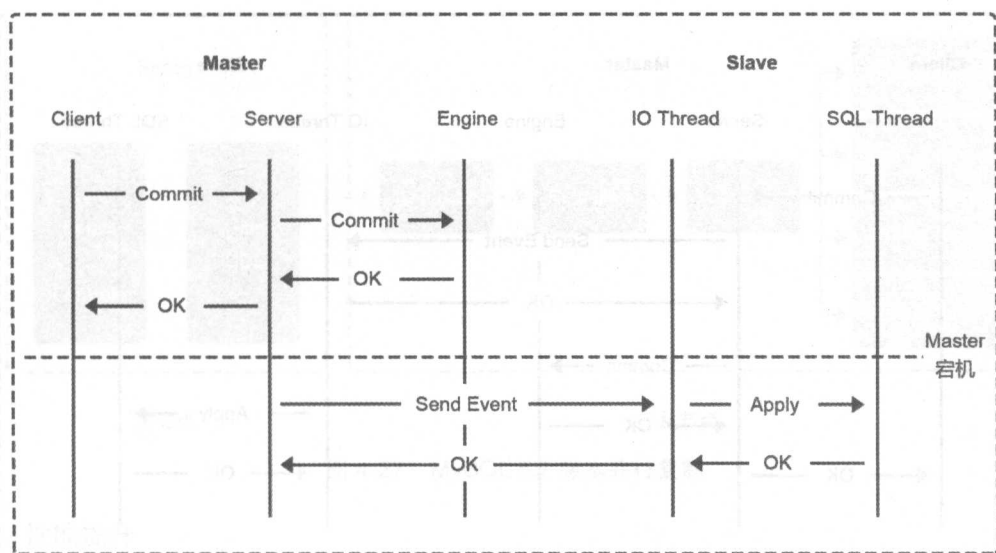


图 4-25 MySQL 异步复制

MySQL 5.5 版本推出了 Semi-Replication 插件，将主节点上的事务提交过程进行了微调，在主节点上提交事务以后，必须等从机收到二进制日志的确认后，主节点再返回给客户端事务提交成功。虽然这样可以确保主从切换后，客户端确认成功提交的事务从节点上一定都有，但是由于主节点上已经提交，还没有传递给从节点，虽然这部分事务没有返回给客户端提交成功，但是已经对其他事务可见，如果主从切换以后丢失，存在脏读的问题，还是无法彻底解决主从数据一致的问题。

网易公司早在 2012 年，就基于 Semi-Replication 插件对 MySQL 内核进行了改进，将主机的事务提交放在发送给从节点二进制日志的后面，确保所有主节点的事务提交对应的二进制日志都已经写入从节点，这样主从切换后，可以确保主从数据的严格一致，如图 4-26 所示。在 2015 年正式 GA 的 MySQL 5.7 版本中，已经增加了该特性，成为 loss-less replication，通过将 `rpl_semi_sync_master_wait_point` 参数设置为 `after sync` 即可开启。

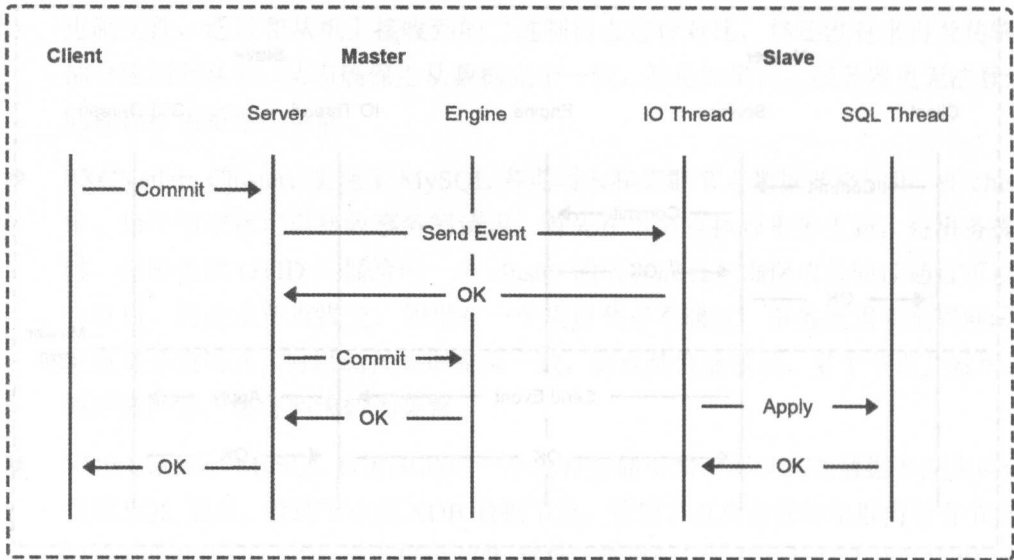


图 4-26 MySQL 同步复制

恢复时间

基于复制实现的 MySQL 高可用故障恢复时间主要包括 3 个环节：主机探测宕机时间、从机二进制日志回放时间和业务切换时间，其中第一和第三部分时间都相对固定且短暂，所以恢复时间实际取决于从机二进制日志的回放时间。由于 MySQL 的二进制日志是在主节点事务执行完成提交过程中产生的，也就是说主节点执行完成以后，从节点才开始执行，所以必然存在主从延迟，同时由于在 MySQL 5.7 版本之前，从机的二进制回放是单线程回放，而主节点是多线程执行，所以主从延迟一直是 MySQL 复制的一个弊病。MySQL 5.7 版本推出了并行复制的特性，如图 4-27 所示，将从机单线程回放改成了多线程回放，通过 `slave_parallel_workers` 设置执行线程数，可以实现多线程并行执行主机上提交的事务。

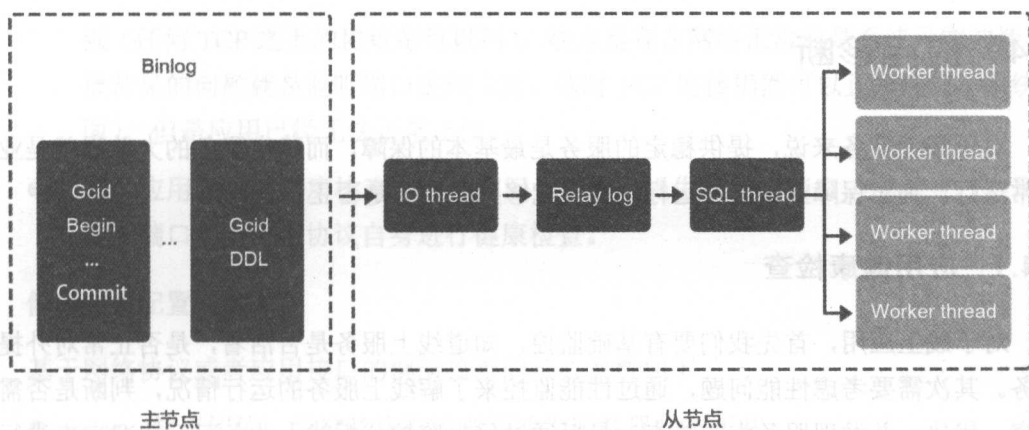


图 4-27 MySQL 5.7 版本并行复制

性能提升

同步复制必然会增加单次事务提交的时间开销，从而对数据库性能产生较大影响。在 MySQL 5.6 版本之前，主机上的事务虽然并行执行，但是在提交阶段，都是串行的。每次事务提交时，都需要和从机确认，MySQL 5.6 版本引入 Group Commit 特性，允许一次提交一组事务，通过增加并发量来弥补响应时间变长的开销。经过测试，基于 Group Commit+ 的同步复制与不带 Group Commit+ 的异步复制性能相近，如图 4-28 所示。

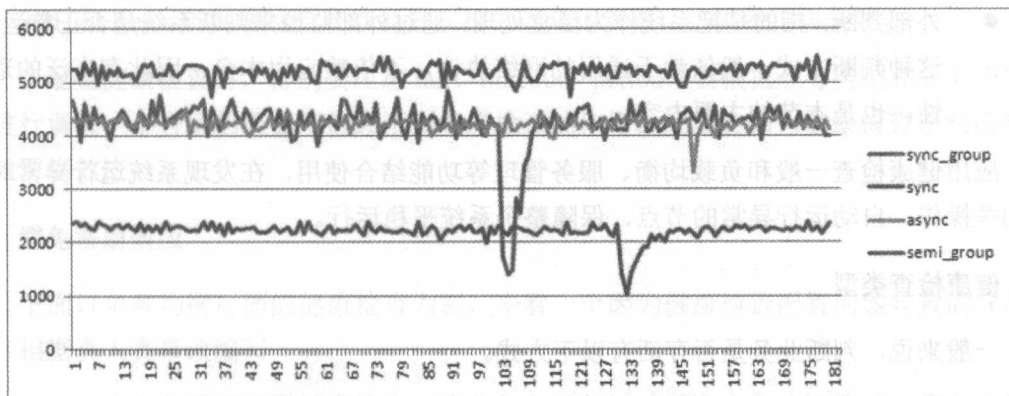


图 4-28 基于 Group Commit+ 同步复制与不带 Group Commit+ 异步复制性能比较

4.4 应用诊断

对于线上业务来说，提供稳定的服务是最基本的保障，而稳定服务的大前提就是业务正常运行。为了保障业务正常运行，就要能够判断业务是否正常工作。

4.4.1 应用健康检查

对于线上应用，首先我们要有基础监控，知道线上服务是否活着，是否正常对外提供服务。其次需要考虑性能问题，通过性能监控来了解线上服务的运行情况，判断是否需要扩容、优化，并发现服务中的热点。最后通过日志监控分析线上业务运行情况，来进行问题的复盘和排查。

基础监控是业务稳定运行的第一重保证！对于一些缺少监控，而且用户反馈渠道不顺畅的业务来说，一个业务挂上几个小时甚至几天都是有可能的。按照判断方式判断业务是否成功，可以分为内部判断和外部判断。

- **内部判断**：指的是在程序内部植入监控模块，通过检查应用内部运行情况来获取应用的运行状态。应用开发的数据一般比较准确，可以进行细致地检查。但是与具体应用相关，缺乏广泛的适用性，我们在此不再展开介绍。
- **外部判断**：指的是把系统作为黑盒处理，通过外部监控来判断系统是否正常运行。这种判断方式一般依赖于通用的网络协议，不依赖应用本身，因此有广泛的适用性，也是本节的主要内容。

应用健康检查一般和负载均衡、服务管理等功能结合使用，在发现系统运行异常时进行相关操作，自动运行异常的节点，保障整个系统平稳运行。

健康检查类型

一般来说，判断业务是否存活有以下方式。

- **进程监控**：这是最直接的监控方式。依赖操作系统实现，只能判断进程是否存活，是较基础的判断。
- **基于网络协议的健康检查**：直接使用网络协议来判断服务是否正常。因为需要明确到进程，最常用的做法就是 TCP 健康检查。这种做法好处是实现简单，适用性

强（任何 TCP 之上的协议都可以用），缺点是存在网络正常但是系统异常的情况。最常见的问题就是监听端口进程卡死，这时 TCP 连接仍然可以正常建立（系统层面），但是应用已经无法正常工作。

- 基于应用协议的健康检查：一般使用业务对外提供的只读接口，或者专门实现的健康端口，对应用协议自身进行健康检查。

健康检查配置参数

基于网络协议或者应用接口的健康检查的核心参数如下。

- rise: 健康阈值，经过该次数检查后认为服务器状态正常。
- fall: 不健康阈值，经过该次失败后认为服务器宕机。
- timeout: 请求超时。
- period: 请求间隔。
- port: 健康检查端口号。
- protocol (可选): 请求协议。
- rl (可选): 请求的 URL。

SQL 和 Redis 的健康检查还涉及协议相关的具体参数，这里就不再赘述。

在配置健康检查时，特别要注意 fall、timeout、period，要根据业务类型和业务响应时间进行调整。不合适的设置不但起不到健康检查的作用，反而会因为健康检查误判而导致系统雪崩效应。

避免雪崩效应

下面以负载均衡后面的健康检查为例，来看一个因为健康检查配置问题导致的异常场景（根据真人真事改编）。

- 一个业务需要配置健康检查，该业务在低负载时平均响应时间是 3s，偶尔会超过 5s，而在系统压力较高时（负载 50%），平均响应时间会达到 6s，随着系统压力增加，响应时间还会进一步延长。
- 系统配置人员在系统上线前测试了健康检查接口，得出响应时间 3s 的结论，因

此在设置健康检查时，timeout 设置为 5s，其他的按照默认参数设置，最终参数如下。

rise: 2

fall: 2

timeout: 5000ms

period: 10000ms

port: 8080

protocol (可选): http

url (可选): /getStatus

- 在线上运行时，偶尔会有超时的情况，但是并没有连续 2 次触发异常。由于有多个节点，偶尔去除 1 个节点也会快速拉起，不会影响用户业务。
- 产品方发起了一次业务推广活动，导致系统压力增加到原来的一倍。运维人员评估任务压力增加后也只是到负载一半，因此没有进行扩容就开展了活动。
- 当线上压力增加时，首先是压力最大的节点超时，从转发列表中移除。接着流量被分担到其他节点上，其他节点压力更大，导致继续移出。直到所有节点都被拿掉，业务完全崩溃。在这个流程中，最恐怖的是，节点被移除的过程，都属于系统正常逻辑，不会触发报警。

在移除的过程中，开始只是业务变慢，并没有异常发生。只有当剩下的集群负载过高导致请求失败时，才会发现问题。

这个问题一旦发生，在入口流量下降之前，系统就无法正常工作。因为移出的节点会因为压力降低而恢复正常，恢复正常后又导入流量，然后负载过高又发生异常，给用户的体验就是产品不稳定、响应变慢和时断时续。规避这种情况可以从以下角度着手。

- 设置超时时间需要考虑到系统在高负载情况下的实际响应能力，提供的接口响应时间需要保证。
- 缓慢宕机，快速拉起，保证压力过高的时候不会过于频繁地关闭节点，同时在节点恢复时可以快速扩充服务。

- 使用弹性伸缩服务，保证后面健康的实例数量，在系统负载过高时自动拉起新的节点来降低负载。

配置实践

下面以网易云基础服务为例，提供健康检查的配置实践。

- 负载均衡提供两种检查模式，是在负载均衡服务中直接对后端进行检查，如图 4-29 所示。

负载均衡服务可以直接在对应监听中配置健康检查参数，也可以自定义健康检查参数。异常的节点可以直接移出转发集群。

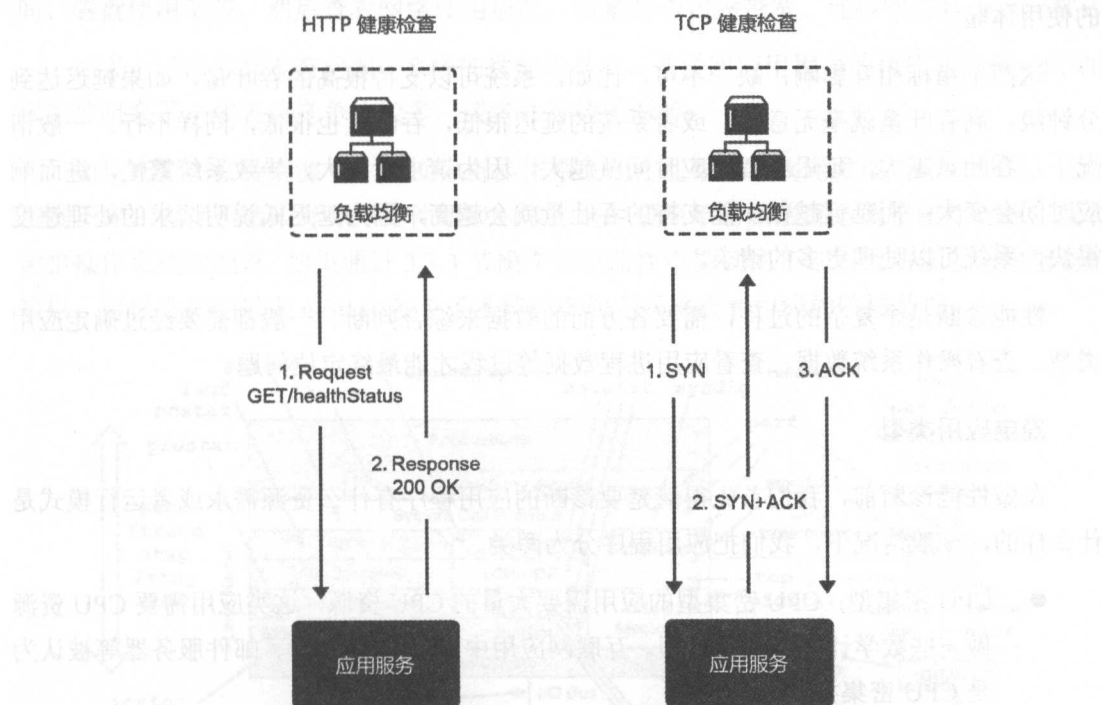


图 4-29 负载均衡提供的两种健康检查方式

● APM

APM 本身提供了对请求的详细分析，通过 APM 提供完善的健康检查功能，可以直接设置报警，发现系统异常。

4.4.2 性能问题诊断

在性能诊断之前，我们要先清楚如何判定，或者说如何确定应用有性能问题，否则无法定位性能问题。总的来说，性能指标主要有以下两点。

- 吞吐量：每秒可以处理的请求数据或者任务数据。
- 响应时间：处理一个请求/任务的时间或者延迟。

整个系统的性能基本由这两个指标来反映，系统对性能指标可能有不同的偏好，在有些场景下，系统可能偏好更高的吞吐量，这样可以同时服务更多的用户；但在另外一些场景下，系统可能偏好低延迟或者响应时间短，这样可以快速返回单个请求，保证用户良好的使用体验。

这两个指标相互影响，缺一不可。比如，系统可以支持很高的吞吐量，如果延迟达到分钟级，高吞吐量就毫无意义；或者系统的延迟很低，吞吐量也很低，同样不行。一般情况下，吞吐量越大，延迟或者响应时间就越大，因为请求量很大，导致系统繁忙，进而响应时间会变大；而延迟越低，能支持的吞吐量就会越高，因为延迟低说明请求的处理速度很快，系统可以处理更多的请求。

性能诊断是个复杂的过程，需要各方面的数据来综合判断。一般都需要经过确定应用类型、查看操作系统数据、查看应用进程数据等过程才能最终定位问题。

确定应用类型

在做性能诊断前，我们首先得清楚要诊断的应用程序有什么资源需求或者运行模式是什么样的，一般情况下，我们把应用程序分为两类。

- CPU 密集型：CPU 密集型的应用需要大量的 CPU 资源，这类应用需要 CPU 资源做一些数学计算或者批处理。互联网应用中的 Web 服务器、邮件服务器等被认为是 CPU 密集型应用。
- I/O 密集型：I/O 密集型应用需要用到大量的内存及底层存储系统，主要是因为它需要处理大量数据。I/O 密集型应用一般不需要 CPU 或者网络（云硬盘之类的存储系统还是需要网络的），它使用 CPU 主要用于发起 I/O 请求，然后处理等待状态。数据库应用一般被认为是 I/O 密集型应用。

这里的 CPU 密集型和 I/O 密集型应用并不是说只用到 CPU 或者 I/O，而是指影响应用性能瓶颈的主要是 CPU 和 I/O。确定了应用的类型，也就知道了影响应用的主要因素，在后面的排查过程中就能更有效地解读数据。

查看操作系统数据

发现应用的情况出现问题时，不一定是应用本身有问题，也有可能只是操作系统资源不够。所以我们要观察收集的操作系统数据，分析性能瓶颈是否在操作系统资源上。

首先是 CPU 利用率，如果 CPU 利用率不高，而系统的吞吐量不高或者延迟很高，就说明我们的程序并没有忙于计算，而是忙于 I/O。其次我们可以看一下 I/O 的情况、等待时间、磁盘使用率等。然后查看网络使用情况、流量是否占满带宽、连接状态是否有异常。

如果上述内容都没有问题，系统的性能还是低，就说明应用程序的代码有问题，比如，程序被阻塞了，或者锁竞争比较多，或者在等待资源等。

如果瓶颈在系统资源上，通过分析操作系统的性能数据，就能发现具体的原因，比如带宽不够，内存不够或者打开文件数限制等。这时最简单的做法就是调整硬件资源，或者调整操作系统的配置。如果通过 3.3.3 节操作系统监控中提到的工具还无法获取到你想要的数

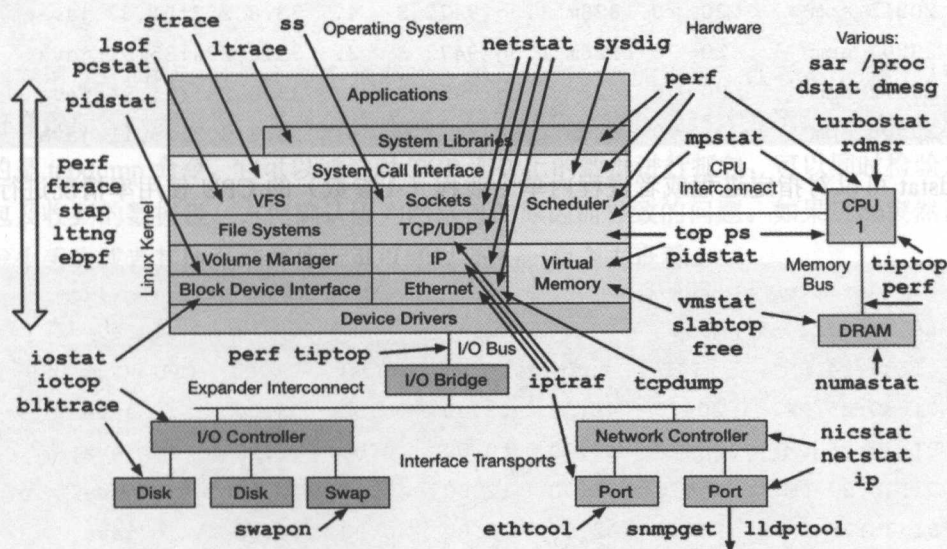


图 4-30 Linux 性能监控工具

查看应用进程数据

如果操作系统本身的负载不高或者资源充足，我们就需要观察应用程序的性能指标。观察应用进程状态，首先要做的就是先看看进程使用的系统资源，数据的获取可以参考之前的应用进程监控。进程的 CPU 使用率、内存使用量、I/O 读写情况都有可能是问题线索。

CPU

当进程的 CPU 指标异常时，如 CPU 使用率很高或很低，与之前的预期不一致，就需要查看具体进程的 CPU 使用情况，或者进程内各个线程的运行情况，有可能因为某个线程出问题（死锁、资源等待等），导致整体进程状态异常。top 及 pidstat 都可以给出某个进程的 CPU 使用情况。使用合适的参数，还可以观察进程中线程的 CPU 使用情况。

```
$ top -H -p 20478
top - 13:24:10 up 130 days, 21:15, 3 users, load average: 0.77, 0.83, 0.86
Threads: 47 total, 0 running, 47 sleeping, 0 stopped, 0 zombie
%Cpu(s): 7.5 us, 2.5 sy, 0.0 ni, 89.0 id, 0.0 wa, 0.0 hi, 1.0 si, 0.0 st
KiB Mem: 4061348 total, 3872328 used, 189020 free, 154572 buffers
KiB Swap: 0 total, 0 used, 0 free, 1440360 cached
  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM   TIME+  COMMAND
 20515 apm       20   0 1828m 1.3g 9472  S   4.7  33.8 257:51.37 java
 20507 apm       20   0 1828m 1.3g 9472  S   3.3  33.8 294:35.91 java
 20512 apm       20   0 1828m 1.3g 9472  S   3.3  33.8 281:41.84 java
 20506 apm       20   0 1828m 1.3g 9472  S   3.0  33.8 303:23.11 java
```

pidstat 可以对指定进程或者进程内单个线程（-t 参数）的 CPU 使用率情况进行定期统计。

```
$ pidstat -u -p 20478 1
Linux 3.2.0-4-amd64      12/18/2016      _x86_64_      (4 CPU)
01:30:24 PM          PID    %usr %system %guest    %CPU   CPU  Command
01:30:25 PM      20478    46.00   12.00    0.00   58.00    0  java
01:30:26 PM      20478    32.00   10.00    0.00   42.00    0  java
01:30:27 PM      20478    35.00   12.00    0.00   47.00    0  java
01:30:28 PM      20478    32.00   12.00    0.00   44.00    0  java
01:30:29 PM      20478    33.00   15.00    0.00   48.00    0  java
```



```
01:30:30 PM      20478    27.00    13.00     0.00    40.00     0 java
01:30:31 PM      20478    34.00    14.00     0.00    48.00     0 java
```

磁盘

如果进程因为等待磁盘操作而导致性能下降，可以观察某个进程的 I/O 读写情况。
pidstat 可以对进程的 I/O 读写情况进行统计。

```
$ pidstat -d -p 20478 1
Linux 3.2.0-4-amd64      12/18/2016      _x86_64_      (4 CPU)
01:46:47 PM      PID    kB_rd/s    kB_wr/s    kB_ccwr/s  Command
01:46:48 PM      20478      0.00      4.00      0.00    java
01:46:49 PM      20478      0.00     40.00      0.00    java
01:46:50 PM      20478      0.00     16.00      0.00    java
01:46:51 PM      20478      0.00    280.00      0.00    java
01:46:52 PM      20478      0.00    800.00      0.00    java
```

在一些场景下，你可能需要知道目标进程是否打开了某个文件（不仅限于磁盘文件，Linux 世界中，一切都是文件，如网络连接、硬件设置等）或者目标进程写日志的目录等，可以通过 lsof 来达到目的。

网络

当应用进程的网络行为不符合预期时，如无法接受新的连接，或者某个连接发送的数据不符合预期等，就需要对具体的连接或者端口的数据进行观察才能确认问题。一般比较常用的是 tcpdump 命令，它可以通过对 TCP 连接上的数据进行抓包，可以同时检验发出的数据包及收到的数据包，方便确认因为网络包的原因而导致的问题。如果进程突然消失、网络发生丢失或者其他系统事件，可以通过 dmesg 命令去查看。

```
$ dmesg | tail
[31399273.118745] [ pid ]    uid  tgid total_vm      rss cpu oom_adj
oom_score_adj name [31399273.118862] [12285] 1012 12285 3027969 862855 0
0          0 java [...]
[31399273.118871] Out of memory: Kill process 12285 (java) score 851 or
sacrifice child
[31399273.120140] Killed process 12285 (java) total-vm:12111876kB,
anon-rss:3451420kB, file-rss:0kB
```

如上面的输出就提到了一个 Java 进程因为 OOM 而被系统杀死。

Java 应用性能诊断

jstack

jstack 可以提供 Java 进程内部每个线程的运行情况，包括线程是否在运行、线程是否在等待锁或者 I/O 等待。jstack 可以定位到线程堆栈，根据堆栈信息定位到具体的应用代码，这些信息对于判断应用进程的运行状态及判断出问题的代码非常有用。以下代码就是一个简单 jstack 输出。

jmap

当发现程序的内存不足、GC 异常，或者怀疑有内存泄露问题时，可以通过 jmap 来获得运行中 Java 程序堆内存的快照，以检查有哪些影响性能的大对象的创建，或者什么对象的数据较多，以及各种对象占用内存的大小等。

常用的使用方式有两种，一种给出进程内所有对象的统计数据，比如哪个类型，创建了多少个对象，占用了多少内存，并以文本格式输出结果，我们可以很直观地对两次的结果进行对比。

```
$ jmap -histo 7599
```

| | num | #instances | #bytes | class | name |
|----|--------|------------|-----------------------------|-------|------|
| 1: | 196770 | 411056864 | [I | | |
| 2: | 669308 | 348008664 | [C | | |
| 3: | 861868 | 188324208 | [B | | |
| 4: | 333390 | 40006800 | java.net.SocksSocketImpl | | |
| 5: | 743247 | 35675856 | sun.nio.cs.UTF_8\$Encoder | | |
| 6: | 755079 | 30203160 | java.lang.ref.SoftReference | | |

另一种以二进制的形式输出结果，包含更多的信息，比如对象是谁创建的，谁在引用等。生成结果后就可以通过 MAT 之类的工具来做分析。

```
$ jmap -dump:format=b,file=test.bin 7599
Dumping heap to /home/apm/test.bin ...
Heap dump file created
```

jstat

如果没有通过 JVM 参数等方式来输出 GC 的情况,就可以通过 jstat 来实时观察 GC 情况,比如发生的次数和耗时,同时 jstat 也可以实时观察堆内存的使用情况。

```
$ jstat -gc 7599 1000
```

| | S0C | S1C | SOU | S1U | EC | EU | OC | OU | PC | PU |
|-----------|-----------|---------|----------|----------|----------|-----------|-----------|----------|----------|----------|
| YGC | YGCT | FGC | FGCT | GCT | 2880.0 | 2880.0 | 2720.0 | 0.0 | 693248.0 | 692906.2 |
| 1398144.0 | 1139334.5 | 47808.0 | 45198.8 | 473067 | 7522.889 | 23 | 24.989 | 7547.878 | | |
| 2880.0 | 2880.0 | 0.0 | 2848.0 | 693248.0 | 254163.5 | 1398144.0 | 1139366.5 | | | |
| 47808.0 | 45198.8 | 473067 | 7522.904 | 23 | 24.989 | 7547.893 | | | | |
| 2880.0 | 2880.0 | 0.0 | 2848.0 | 693248.0 | 476794.8 | 1398144.0 | 1139366.5 | | | |
| 47808.0 | 45198.8 | 473067 | 7522.904 | 23 | 24.989 | 7547.893 | | | | |
| 2880.0 | 2880.0 | 0.0 | 2848.0 | 693248.0 | 669660.6 | 1398144.0 | 1139366.5 | | | |
| 47808.0 | 45198.8 | 473067 | 7522.904 | 23 | 24.989 | 7547.893 | | | | |
| 2880.0 | 2880.0 | 2736.0 | 0.0 | 693248.0 | 218081.7 | 1398144.0 | 1139414.5 | | | |
| 47808.0 | 45198.8 | 473068 | 7522.921 | 23 | 24.989 | 7547.910 | | | | |
| 2880.0 | 2880.0 | 2736.0 | 0.0 | 693248.0 | 431233.3 | 1398144.0 | 1139414.5 | | | |
| 47808.0 | 45198.8 | 473068 | 7522.921 | 23 | 24.989 | 7547.910 | | | | |
| 2880.0 | 2880.0 | 2736.0 | 0.0 | 693248.0 | 667210.6 | 1398144.0 | 1139414.5 | | | |
| 47808.0 | 45198.8 | 473068 | 7522.921 | 23 | 24.989 | 7547.910 | | | | |
| 2880.0 | 2880.0 | 0.0 | 2752.0 | 693248.0 | 200328.6 | 1398144.0 | 1139482.2 | | | |
| 47808.0 | 45198.8 | 473069 | 7522.935 | 23 | 24.989 | 7547.924 | | | | |
| 2880.0 | 2880.0 | 0.0 | 2752.0 | 693248.0 | 390017.8 | 1398144.0 | 1139482.2 | | | |
| 47808.0 | 45198.8 | 473069 | 7522.935 | 23 | 24.989 | 7547.924 | | | | |
| 2880.0 | 2880.0 | 0.0 | 2752.0 | 693248.0 | 634780.9 | 1398144.0 | 1139482.2 | | | |
| 47808.0 | 45198.8 | 473069 | 7522.935 | 23 | 24.989 | 7547.924 | | | | |

Btrace

JVM 提供的工具一般只能显示 JVM 层面的信息,如果碰到线上问题,需要知道一些代码层面的信息怎么办?我们可能在程序里打一些日志来排查,但每一次查看问题都需要更改代码,重新部署,然后观察。这种方式对于在线应用的排查来说,一方面效率很低,另一方面破坏问题出现的上下文,导致无法重现问题。而 Btrace 就是一个可以在不修改应用代码、不重启进程的情况下,动态地将跟踪字节码注入运行类中,方便查看应用运行信

息的工具。图 4-31 展示了 Btrace 的工作原理。

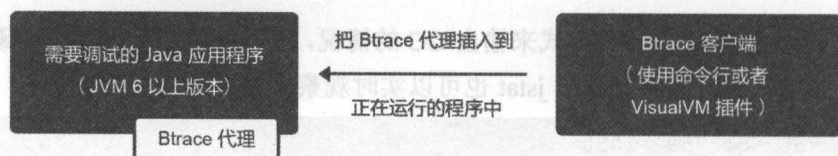


图 4-31 Btrace 的工作原理

具体如何使用 Btrace 可以参考它的用户手册。

4.4.3 基于日志的故障诊断

日志是故障诊断的重要手段，可以记录程序运行时的动态信息，帮助维护人员分析重现错误，进而更正系统错误，提高系统运行的可靠性。但通过日志进行故障诊断并非易事。随着时间的积累，日志越来越庞大，用户常常无法快速地定位问题日志。而且单台机器的空间有限，每隔一段时间就会删除日志文件，这对于问题的追溯也造成了困难。

尤其对于分布式系统来说，日志分散在系统各个节点上，更增加了日志的查看和错误发现难度，用户登录几十或上百台机器查看日志是个极大的负担。另外对于多个服务组成的系统（比如微服务系统）来说，各服务之间的调用关系链也难以跟踪，出现一个错误时要彻查根源非常困难。

本节将对利用日志进行故障诊断的现状和常用方法进行综述，并介绍网易云平台中改进的解决方案，结合网易云平台和日志服务的实际例子进行说明，以便读者理解和使用。

日志诊断现状

对于单机系统来说，目前业内采用最普遍的仍然是 `tail -f + grep/awk` 的方法。或者手动将日志文件从服务器拉取到本地之后再利用工具搜索查看，这些方法简单粗暴，也存在很多问题，比如日志文件过大、难以查找、效率低、权限无法控制等。

大家已逐渐发现上述方法不方便，由此兴起了很多分布式的日志集中管理系统，比如 Splunk、ELK 等。这些系统解决了分布式登录、权限管理、难以查找定位等问题，极大地方便了日志的维护，但对于其系统本身的维护和应用又存在一定的困难。

网易云基础服务是以分布式微服务为基础的平台，研发团队很早就意识到日志对系统

维护的关键作用，并在最早的版本中提供了日志服务，包括日志实时跟踪、日志搜索定位、按时间定位、request_id 关联、日志搜索结果上下文查看等功能。下面以网易云基础服务平台的实际案例来演示如何进行便捷的分布式系统错误日志定位。

实时跟踪

小王的系统已经部署在网易云云基础服务平台，他首先想知道目前系统运行状态是否正常，需要看实时的日志输出。小王的系统是由多个容器组成的微服务系统，他需要逐个登录不同的容器去进行日志查看吗？当然不用，因为日志服务已经默默地在后端做好了一切。小王只需要打开日志服务页面，选择好相应的服务名称，相关的日志就会实时输出到页面中，如图 4-32 所示。

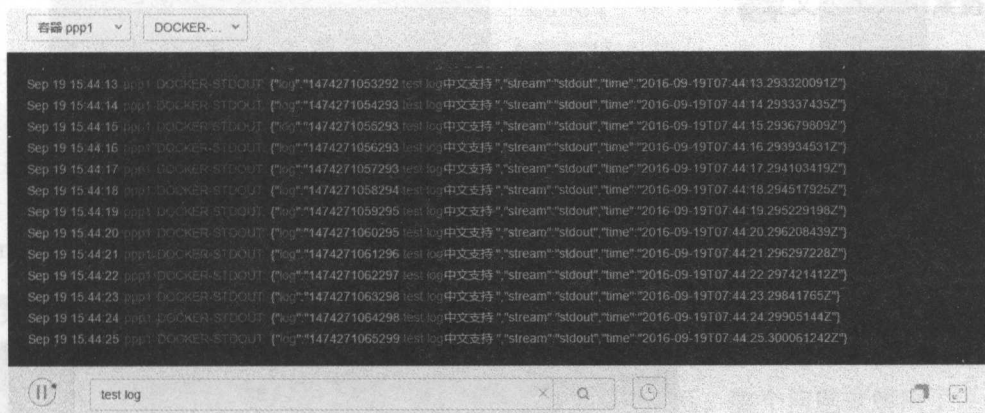


图 4-32 通过日志服务查看日志

小王观察了一会日志，发现一切正常，基本上就放心了。但由于日志比较多，页面刷新得比较快，有些细节无法查看清楚，是否可以只根据某些关键词过滤输出呢？

查找定位

小王想要根据关键词过滤实时输出，在日志服务中，这其实也非常简单。只要将相关的关键词输入到搜索框就可以（支持标准 Lucene 搜索语法）。在搜索过程中，包含关键词的日志会实时刷新出来。如果想要查看之前包含关键词的日志，只要鼠标往上滚动即可，如图 4-33 所示。

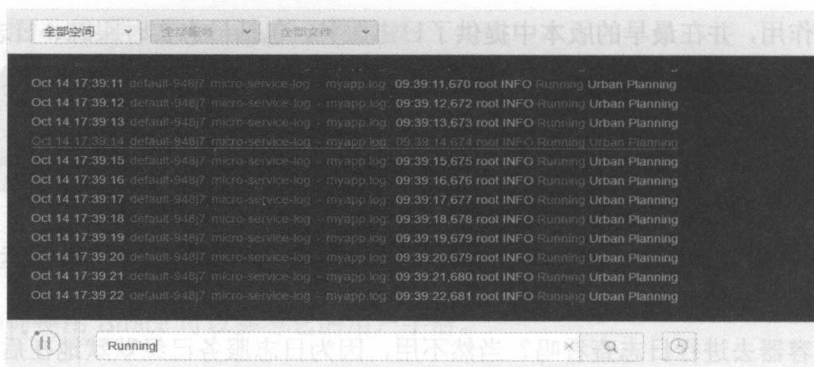


图 4-33 日志服务支持根据关键词过滤进行实时输出

搜索结果上下文查看

小王用一个关键词搜索了日志之后，所有相关的日志都列出来了。可是小王想要查看错误的具体原因，还需要错误日志的上下文，仅仅显示错误日志内容是不够的。日志服务能解决该问题吗？答案也是肯定的，日志服务独创地开发了上下文查看功能，小王要查看前后因果日志，只要在错误日志上点击，日志服务就会弹出日志上下文的查看窗口，如图 4-34 所示。鼠标上下滚动，还可以查看无限内容的日志。

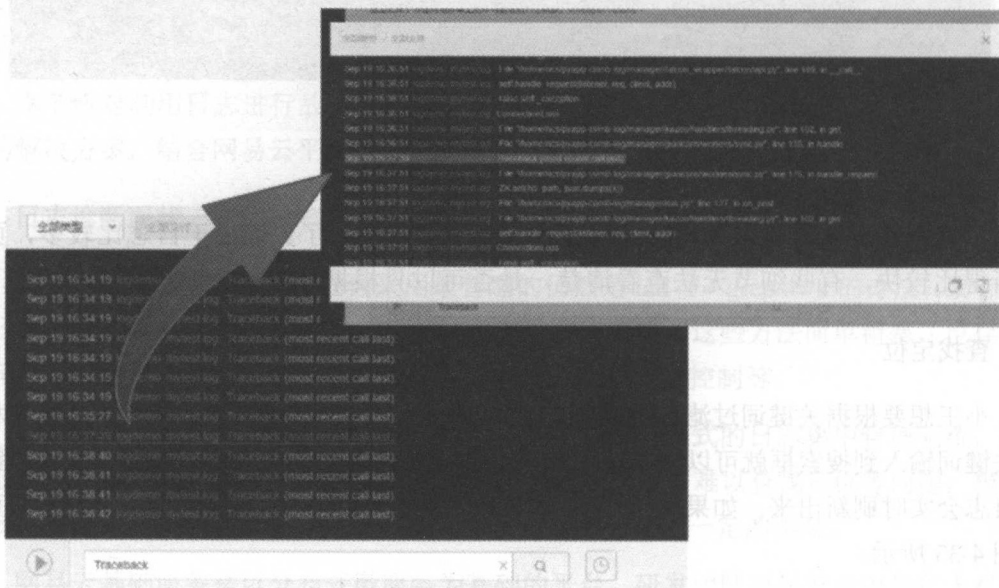


图 4-34 搜索结果上下文查看

按时间定位

系统已经跑了一段时间了。有一天晚上，系统偶尔出现了点异常。第二天，小王想要查看当时那个点的日志，需要从最新的日志开始一直往后翻吗？不用。日志服务提供的按时间定位日志功能，如图 4-35 所示，只要输入具体的时间点，就能准确定位到时间点附近的日志，方便定位查找，而且系统马上就返回结果，小王可以快速查看问题具体原因。

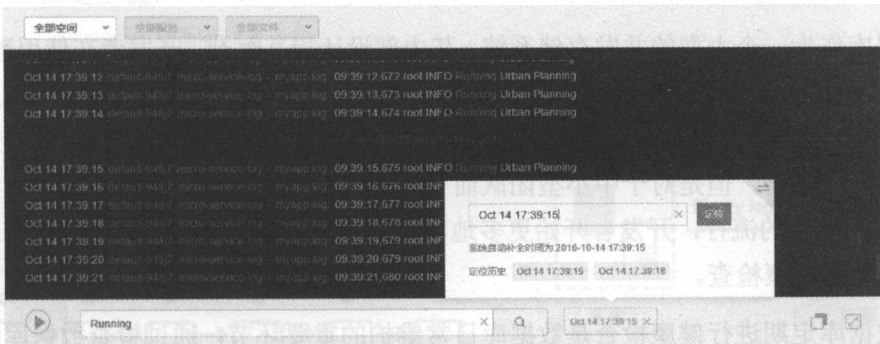


图 4-35 按时间定位日志

request_id 关联

系统里有很多个模块，也跑很多线程。一个用户的请求，需要很多的处理流程，可系统打出来的日志是杂乱无章的，并不能保证每个事件点之间的顺序。这让小王在追溯整个事件时很受伤。自从小王给每个请求都加上了 request_id 之后，整个问题就解决了，借助日志服务的搜索查看及日志统一管理功能，小王只要输入 request_id，就可以清晰地看到整个事件的处理流程，如图 4-36 所示。

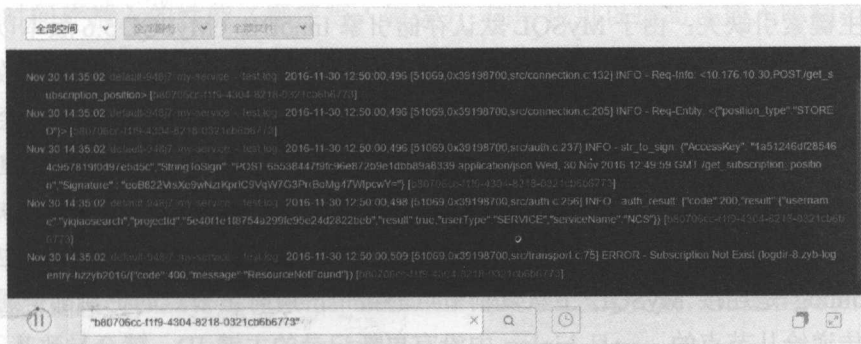


图 4-36 通过 request_id 关联查看事件处理流程

就这样，通过使用网易云基础服务中日志服务的主要功能，小王很好地克服了分布式系统的日志故障诊断难题，此外日志服务还支持用户自定义订阅模式，可实现支持日志监控、报警、日志加工等服务。

4.5 数据库诊断

数据库作为一个大型的并发存储系统，其内部设计极其复杂，开发者在使用数据库时，面临着可用性、可靠性、性能、安全、扩展性等多重挑战，这使得数据库的使用具有较高的技术门槛。一般大型团队都会雇佣专职的 DBA 来维护数据库的日常运行，指导开发者建立正确的使用姿势，但是对于中小型团队而言，受限于人力成本，这种模式难以实现。随着 DevOps 理念的流行，开发者开始更多地参与和承担数据库的运维工作，其中就包括对数据库的定期健康检查。

对数据库定期进行健康检查是数据库日常维护的重要环节，通过检查数据库的各项运行指标，评估系统的运行风险，提前将风险消灭在摇篮中，能够有效提高数据库服务的质量。数据库的健康检查涉及索引设计、容量规划、服务安全、参数配置、用户访问、集群复制 6 个方面。

索引设计

合理的索引设计能够有效加速数据库的访问，提高查询的执行效率，减少用户查询对服务端的资源消耗。但是不合理、低效、冗余甚至无效的索引不仅无法起到加速查询的效果，反而会影响数据库的插入、更新性能，甚至是数据库的高可用方案能否生效。

- **主键索引缺失：**由于 MySQL 默认存储引擎 InnoDB（MySQL 5.6 版本以上）使用的是聚簇索引表设计，这就要求所有的表必须包含一个主键，所有的数据记录按照主键次序构建 B+树。如果用户在创建表时显式指定主键，数据库就会使用用户指定的主键构建 B+树，但是如果用户没有显式指定主键，同时也没有创建任何唯一键索引，InnoDB 为了确保每张表至少包含一个主键，会默认为用户生成一个“隐含主键”，该主键对用户不可见，甚至对于 MySQL Server 层的 binlog 也不可见。binlog 是连接 MySQL 主从复制节点的纽带，所有主节点的更新都是通过 binlog 传递给从节点的，一旦 binlog 中没有更新记录的主键 ID，就会导致基于 Row 格

式的 binlog 在从节点执行时，无法确定一条唯一记录，只能通过全表扫描来进行匹配，大幅降低了从机的执行效率，造成复制延迟，如图 4-37 所示。如果是高可用故障切换的从节点，会导致切换的时间大幅增加，甚至会导致高可用机制失效。如果是实现读写分离的只读从节点，则会导致应用读到的数据可能是很久以前的旧数据。所以使用 InnoDB 存储引擎的 MySQL 用户在创建表时，最好显式指定主键。

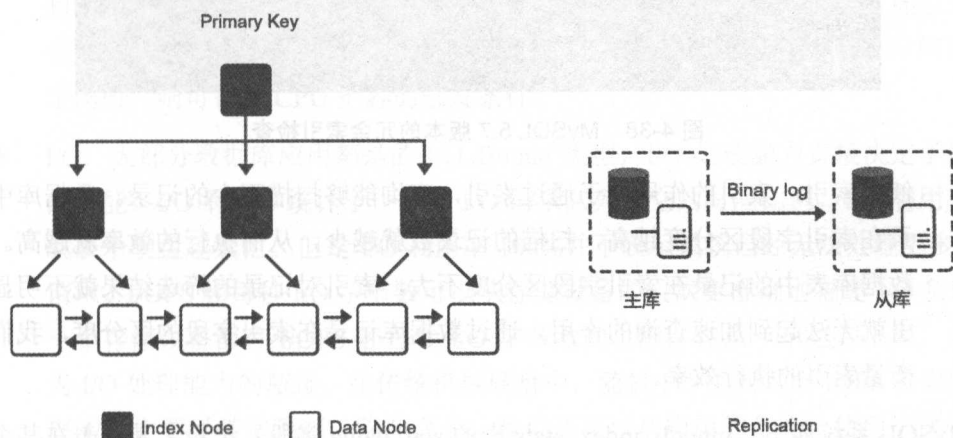


图 4-37 MySQL 的索引和主从同步机制

- 主键索引与业务相关：如果用户在创建表时指定的主键与业务相关，就会被频繁更新，从而引起 MySQL 数据库的 InnoDB 存储引擎进行频繁的节点合并和分裂，造成大量额外的系统 I/O 开销，影响数据库的插入和更新性能。我们推荐开发者在创建表时指定与业务无关的自增字段作为主键，这样不仅能提高按时间序插入的性能（顺序写入硬盘），同时也可以提高按插入时间范围检索的查询效率。
- 冗余索引：如果一个索引涉及的字段属性包含另外一个索引涉及的字段属性，同时两个索引字段顺序一致，且两个索引的首字段属性相同，则可以认为涉及字段少的索引为冗余索引。在 MySQL 5.7 版本推出 sys 库之前，我们可以通过 percona 的工具 pt-duplicate-key-checker 来完成对冗余索引的检查，在 MySQL 5.7 版本中，我们可以通过 sys 库 schema_redundant_indexes 表来完成，如图 4-38 所示。

```
mysql> select * from sys.schema_redundant_indexes \G;
***** 1. Row *****
      table_schema: blog
      table_name: sbtest3
      redundant_index_name: a_4
      redundant_index_columns: b,c
      redundant_index_non_unique: 1
      dominant_index_name: a_2
      dominant_index_columns: b,c,d
      dominant_index_non_unique: 1
      subpart_exists: 0
      sql_drop_index: ALTER TABLE `blog`.`sbtest3` DROP INDEX `a_4`
1 row in set (0.00 sec)

ERROR:
No query specified
```

图 4-38 MySQL 5.7 版本的冗余索引检查

- 低效索引：索引的作用在于通过索引，查询能够扫描更少的记录。数据库中的记录在索引字段区分度越高，扫描的记录数就越少，从而执行的效率就越高。如果数据库表中的记录在索引字段区分度不大，索引对记录的筛选结果就不明显，索引就无法起到加速查询的作用。通过数据库记录在索引字段的区分度，我们可以衡量索引的执行效率。

MySQL 系统库下，`innodb_index_stats` 表的 `stat_value` 字段，记录了某张表在某个索引的不同取值的记录个数，`innodb_table_stats` 表的 `n_rows` 字段记录了某张表的总记录数，二者相除，即可得到数据库记录在某个索引的区分度，越接近 1，表示区分度越高，低于 0.1，则说明区分度较差，开发者应该重新评估 SQL 语句涉及的字段，选择区分度高的多个字段创建索引，通过运行下面的 SQL 语句，就可以计算每张表的索引区分度。

```
select t.database_name AS db , t.TABLE_NAME AS table , i.INDEX_NAME AS
index name , i.stat_description as cols , i.stat_value AS differRows, t.n_rows as
rows, ROUND(((i.stat_value / IFNULL(IF(t.n_rows < i.stat_value, i.stat_value,
t.n_rows), 0.01))), 2) AS sel_percent from mysql.innodb_index_stats i INNER JOIN
mysql.innodb_table_stats t on i.database_name = t.database_name and i.table_name=
t.table_name where t.table_name= % AND t.database_name= % AND i.INDEX_NAME !=
'PRIMARY' and i.stat_name like '%n_diff_pfx%';
```

- 无效索引：如果一个索引始终无法被查询使用，它的存在就只能增加数据库的维护开销，开发者应该及时删除这些索引。通过 MySQL 5.7 版本 `sys` 库 `schema_unused_indexes` 视图，可以查看当前实例有哪些索引从未被使用。

容量规划

数据库的运行依赖计算、存储、网络等多种资源，通过对各种资源的使用情况分析，对资源进行合理的规划配置，是数据库稳定运行的必要条件。

- CPU：通常使用 CPU 利用率衡量 CPU 的繁忙程度，通过 `top` 命令，开发者可以查看 CPU 利用率实时变化。CPU 利用率持续超过 80%，预示计算资源已经接近饱和，如果开发者已经做过 SQL 优化，就需要使用更高配置的 CPU。通过查看 7 天内 CPU 利用率超过 80% 的时间占整体时间的百分比，以及单次持续时间超过一定阈值，则可视为 CPU 扩容的触发条件。
- I/O：大部分数据库应用都是的 I/O Bound 类型，I/O 处理能力直接决定了数据库的性能。I/O 利用率统计了一秒内 I/O 请求队列非空的时间比例，I/O 利用率越高就表示硬盘越繁忙。但是 I/O 利用率 100% 并不表示系统已经无法处理更多的 I/O 请求。I/OPS 和每秒 I/O 字节数可以从存储设备的层次更准确地描述 I/O 负载。每一个存储设备都有 I/OPS 和每秒 I/O 字节数的上限，任意一个达到上限，就会成为 I/O 处理能力的瓶颈，在传统机械硬盘中，随机 I/O 主要受到 IOPS 的限制，顺序 I/O 主要受带宽限制。除此之外，我们还可以从应用的角度，使用一次 I/O 请求的响应时间来描述 I/O 负载，一次 I/O 请求的响应时间包括其在队列中的等待时间和实际 I/O 处理时间之和。开发者可以通过 `iostats` 很方便地收集这些数据。如果这些指标在一段时间内持续接近设定上限，就可以认为 I/O 过载，通过扩大内存，让更多的读写请求命中缓存以缓解硬盘 I/O。另外，使用更高配置的存储设备，例如固态硬盘，也可以大幅提高系统的 I/O 处理能力。
- 存储空间：存储空间不足会导致严重的系统故障，数据库可能会宕机，更为严重的是数据库进程存活，但是无法响应服务，从而造成基于进程的宕机监控失效。根据 7 天内数据库中存储数据的变化，我们可以按照一定的拟合算法，估算出未来 3 天内数据的增长情况，来判断实例是否存在存储空间不足的风险。
- 内存：使用 InnoDB 存储引擎的 MySQL 数据库在实例启动时，就会预分配一块固定大小的内存空间，所有读写请求都会在该空间中完成，如果内存中缓存了用户读写的数据，则直接读取内存，如果内存中没有用户读写的数据，则需要将数据先从硬盘中装载进内存中，由于内存的读写速度远远快于硬盘，这就使得读写请求是否命中内存决定了读写请求的处理速度。内存空间越大，缓存数据越多，命

中的概率也就越大。所以我们可以使用缓存命中率来衡量内存空间大小是否满足应用的需求。在 MySQL 中，show engine innodb status 命令的 Buffer pool hit rate 可以度量近一段时间范围内 Buffer pool 的命中情况，如图 4-39 所示。

```

BUFFER POOL AND MEMORY
-----
Total memory allocated 10554130432; in additional pool allocated 0
Dictionary memory allocated 4028920
Buffer pool size 629120
Free buffers 0
Database pages 1046151
Old database pages 386156
Modified db pages 13
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 405083400, not young 0
17.32 youngs/s, 0.00 non-youngs/s
Pages read 329348336, created 562224, written 499269328
28.18 reads/s, 0.06 creates/s, 132.94 writes/s
Buffer pool hit rate 996 / 1000, young-making rate 0 / 1000 not 0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 1046151, unzip_LRU len: 104616
1/0 sum[6492]:cur[794], unzip sum[5031]:cur[27]
Async Flush: 0, Sync Flush: 0, LRU List Flush: 0, Flush List Flush: 499269328

```

图 4-39 度量近一段时间范围内 Buffer pool 的命中情况

- 网络：网络带宽在数据库返回记录较多的情况下，也可能会成为系统的瓶颈。一般我们使用每秒网络流入和流出字节数来衡量网络流量是否达到带宽限制。在云环境下，每台虚拟机或者容器都有一定的网络带宽配额，私有网络的配额相对比较大，公网配额与用户付费相关；使用 iftop 可以查看当前系统的网络流量。

服务安全

- 弱密码：MySQL 的登录认证使用 IP 和账户密码的方式，很多开发者为了方便记忆，习惯将数据库密码设置为弱密码，这非常危险。数据库中的数据很多涉及敏感业务，弱密码容易被破解，对数据库中的数据来说是一个严重安全隐患。MySQL 系统库下 user 表的 password 字段保存了所有用户的密码，MySQL 使用 2 次 sha-1 的不可逆加密算法，所以我们无法通过 password 字段获取用户的密码内容，但是我们可以通过将常见弱密码制成彩虹表，模拟 MySQL 的加密算法，匹配 password 字段，即可发现数据库中的弱密码账号。
- 网络安全：在一般的业务架构中，数据库都不会直接服务于终端用户，而是服务于运行业务逻辑的应用程序。所以数据库和业务程序之间出于安全的考虑，会选

择使用私有网络。即便如此，为了避免数据库连错，也需要在设置数据库账号时，增加 IP 来源限制。在一些特定的场景下，如果数据访问必须借助公网来实现，就会将数据库暴露在公网上。使用公网数据库实例，必须要配置防火墙，否则存在被攻击的隐患。通过 iptables 我们可以控制访问数据库的来源 IP。

- 权限检查：MySQL 提供了多种权限配置，为了方便管理及避免误操作，一般会将管理权限和访问权限配置成两个不同的账号，禁止使用管理权限作为业务程序访问数据库的账号。通过系统库 MySQL 库的 user 表可以确认各个账号拥有的权限，尽量避免业务账号拥有 super 权限。

参数配置

- 内存相关参数：MySQL 数据库的内存使用包括共享内存与连接独占内存两个部分。每一个用户新建连接，数据库都要分配一块固定大小的内存空间保存用户的临时数据，这些空间为单个连接独占。在 MySQL 实例启动时，系统同时也会预先分配一些实例级别的共享内存空间，例如 Innodb_buffer_pool、Innodb_log_buffer_pool 等，供所有连接共享。独占内存空间乘以最大连接数加上共享内存空间，我们可以计算出 MySQL 最大可使用的内存空间，如果超过实际物理内存大小，就存在 MySQL 进程被 Linux 操作系统强行 oom kill 风险，导致实例宕机。MySQL 的这些内存空间都可以通过配置参数指定大小，如果超过实际内存空间，应该调整相应参数配置，最常见的是调整 Innodb_buffer_pool 和最大连接数。

```
key_buffer_size + query_cache_size + tmp_table_size +  
innodb_buffer_pool_size + innodb_additional_mem_pool_size + innod  
b_log_buffer_size + max_connections * (sort_buffer_size +  
read_buffer_size + read_rnd_buffer_size + join_buffer_size + t  
hread_stack + binlog_cache_size)。
```

- Innodb 日志相关参数：innodb_log_file_size 定义了 Innodb 重做日志的大小，如果参数设置过小，会导致数据库写操作频繁卡顿，如果设置过大，会导致数据库实例重启或者故障恢复花费大量的时间。对于使用固态硬盘等高配置存储设备的数据库，可以将重做日志设置大一些，对于使用机械硬盘的数据库，应该设置小一些，一般在 512M 到 4G 之间。innodb_flush_log_at_trx_commit 定义了重做日志的刷新节奏，如果该参数非 1，会导致数据库宕机重启后丢失部分更新数据，对于

数据可靠性要求较高的应用造成严重影响。

- 二进制日志相关参数：binlog 主要用于 MySQL 集群复制及故障恢复担任协调者的作用。binlog_format 定义了 binlog 的格式，主要包括 ROW、STATEMENT 和 MIXED 3 种格式，ROW 格式是最安全的一种日志格式，会保证主从数据的严格一致，建议开发者选用 ROW 格式。但是 ROW 格式的 binlog 会占用更多的存储空间，通过 expire_logs_days 可以控制保存 binlog 的天数，如果 binlog 占用的存储空间比例超过 50%，就要考虑适当减少 binlog 的保存天数。sync_binlog 参数定义了 binlog 刷新硬盘的节奏，如果非 1，就会导致宕机重启后最近的更新数据丢失。
- 连接数相关参数：MySQL 有最大连接数限制 max_connections，如果应用连接超过 max_connections 限制，就会得到 out of max connections 异常，无法建立连接。show processlist 可以查看当前的连接数，如果接近最大限制，就存在无法新建连接的风险。通过在应用端使用连接池可以控制数据库的连接数。

用户访问

- 慢连接：慢查询数量是最直观的反映数据库处理能力是否满足业务需求的指标。通过设置 slow_query_log 可以开启慢查询日志，MySQL 数据库会将执行时间超过 long_query_time 的查询记入慢查询日志，如果某个时间段内，慢查询数量急剧增加，开发者就必须关注数据库的性能问题，首先需要进行 SQL 优化，其次考虑资源是否需要扩容，最后可能需要数据库水平扩展方案，包括创建只读从节点。
- 死锁数量：两个事务涉及的数据库记录有重叠，如果 SQL 语句的加锁顺序不一致，就会导致事务之间的死锁。虽然 MySQL 数据库会自动检测死锁并强制回滚系统认为代价较小的事务，但是死锁的检测与事务回滚都有较大的代价，会严重拖慢数据库的性能，所以当系统中出现大量死锁时，开发者必须重视，要分析发生死锁的事务 SQL 语句的加锁规则、调整 SQL 语句。通过 show engine innodb status 可以查看死锁的相关信息及系统的处理过程，如图 4-40 所示。

```

LATEST DETECTED DEADLOCK
*****
151224 17:41:48
*** (1) TRANSACTION:
TRANSACTION 128BC9F, ACTIVE 10 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 376, 2 row lock(s)
, undo log entries 1
MySQL thread id 1889006, OS thread handle 0x7f939269a700, query id 56466611 localhost rdsadmin Updating
update user set account = 100 where id =5
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 16 page no 3 n bits 80 index 'PRIMARY' of table 'cloud_study`.`user` trx id 128B
Record lock, heap no 8 PHYSICAL RECORD: n_fields 6; compact format; info bits 0
 0: len 4; hex 80000005; asc      ;;
 1: len 6; hex 00000128bc9e; asc      (  ;;
 2: len 7; hex 68000002172f88; asc h      /  ;;
 3: len 8; hex 5a68616e6753616e; asc ZhangSan;;
 4: len 7; hex 4e657465617365; asc Netease;;
 5: len 4; hex 80000000; asc      ;;

*** (2) TRANSACTION:
TRANSACTION 128BC9E, ACTIVE 44 sec starting index read
mysql tables in use 1, locked 1
3 lock struct(s), heap size 376, 2 row lock(s)
, undo log entries 1
MySQL thread id 1889173, OS thread handle 0x7f93924d3700, query id 56466637 localhost rdsadmin Updating
update user set corp =
Netease' where id = 6
(2) HOLDS THE LOCK(S).
RECORD LOCKS space id 16 page no 3 n bits 80 index 'PRIMARY' of table 'cloud_study`.`user` trx id 128B
Record lock, heap no 8 PHYSICAL RECORD: n_fields 6; compact format; info bits 0
 0: len 4; hex 80000005; asc      ;;
 1: len 6; hex 00000128bc9e; asc      (  ;;
 2: len 7; hex 68000002172f88; asc h      /  ;;

```

图 4-40 数据库中的死锁信息

集群复制

- **数据安全：**复制是 MySQL 多个节点之间实现数据同步的重要机制，主要用于搭建高可用实例主从节点及提供多个只读从节点提高读扩展能力。节点之间的数据是否最终一致对于高可用方案是否生效、只读实例读取的数据是否正确有着严重影响。从机执行 `show slave status` 可以获取从机的复制状态，`Slave_IO_Running` 和 `Slave_SQL_Running` 分别表示 I/O 和 SQL 线程是否正常运行，如果不正常，就要及时处理。参数 `relay_log_recovery` 和 `relay_log_info_repository` 影响从节点宕机重启后，与主机的复制位置是否正确，位置错误可能导致数据错误。
- **复制性能：**复制延迟经常用来评估复制性能是否满足业务需求。`Show slave status` 的 `Seconds behind master` 字段标识了从机落后主机的延迟时间。如果延迟较长，就会影响高可用实例主从切换的时间及只读从节点是否能够及时读到最新数据。通过使用并行复制技术可以提高从节点的复制性能。MySQL 5.6 版本提供了基于

Database 级别的并行复制, 通过 `slave_parallel_workers` 设置并行线程数; MySQL 5.7 版本提供了基于 `LOGICAL_CLOCK` 的并行复制, 主机上同一个 Group 提交的 binlog 中包含事务在从机并行执行, 相比 database, 具备更高的并发性, 除了设置 `slave_parallel_workers`, 还需要将 `slave-parallel-type` 设置为 `LOGICAL_CLOCK`。`slave_preserve_commit_order=1` 可以确保从机并行执行的事务按序提交。同时从机的 `log_bin` 和 `log_slave_updates` 参数必须同时开启。

智能数据库健康诊断系统

使用网易云基础服务的开发者, 可以使用平台提供的智能健康诊断系统对数据库服务中的关系型数据库实例进行自动健康检查。检查内容覆盖 6 个大类 22 个子项, 检查结束后根据检查结果, 自动生成健康指数。开发者根据健康指数, 可以快速判断系统存在的风险严重程度, 同时平台提供了该分数在所有实例的健康检查中的排名, 如图 4-41 所示。

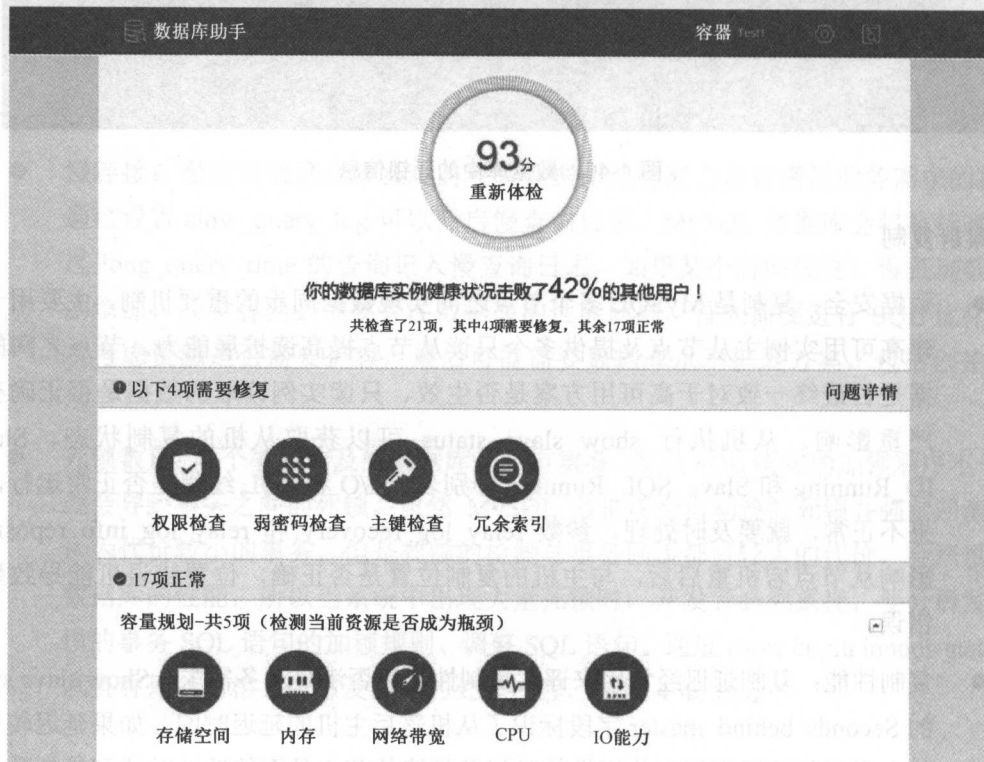


图 4-41 网易云提供的智能数据库健康诊断系统

有风险的项目平台会给出风险提示标识，开发者点击风险项目，会看到系统对该风险的详细描述及相应的修复建议，如图 4-42 所示。针对部分检查内容，系统提供了一键自动修复功能。

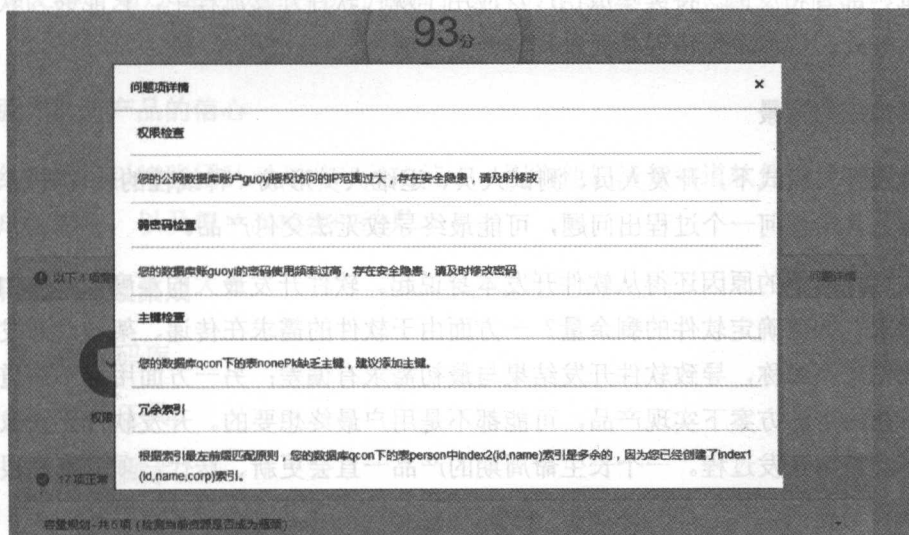


图 4-42 风险描述及修复建议

4.6 DevOps

DevOps 理念是提倡开发、测试、IT 运维之间的高度协同，从而在完成高频率部署的同时，提高生产环境的可靠性、稳定性、弹性和安全性。什么是开发和 IT 运维？因为产品价值流在业务（定义需求）和客户（交付价值）之间。

DevOps 本身并不能完全被工具或软件来简单地定义或量化。但工具或软件是实现 DevOps 的一个重要组成部分，现在流行的 Docker 就是实现 DevOps 最合适的工具之一。Docker 的起源和产品功能，与 DevOps 的理念非常匹配。Docker 完全有能力来加速、保障软件的生命周期。从这几年的行业发展来看，Docker 作为一款工具，的确在帮助企业践行 DevOps 理念，同时也借助这款工具的流行，使 DevOps 在更大的群体中获得推广。

4.6.1 持续集成

持续集成是一种软件项目管理方法，依据资产库（源码、类库等）的变更自动完成编译、测试、部署和反馈。持续集成已广泛应用于现代软件开发流程中，它能够为软件开发带来两大好处：快速发现错误和促进代码分支集成。

持续集成的背景

在传统开发模式下，开发人员、测试人员、运维人员形成一种线性的软件开发、测试、部署流程。其中任何一个过程出问题，可能最终导致无法交付产品。

导致这种问题的原因还得从软件开发本身说起。软件开发最大的难度在于：如何确定软件的需求？如何确定软件的剩余量？一方面由于软件的需求在传递、架构、开发过程中可能造成信息不对称，导致软件开发结果与最初需求有偏差；另一方面用户不知道自己需要什么，在协定的方案下实现产品，可能都不是用户最终想要的。开发软件无法像造汽车一样，看到整个开发过程。一个长生命周期的产品一直会更新、迭代，项目管理根本无法确定软件剩余量。

持续集成通过把一个产品细化为多个功能，逐步实现和完善产品功能，从而改善传统软件流程。Martin Fowler 大师定义持续集成是一种软件开发实践，即团队开发成员经常集成他们的工作，通常每个成员每天集成一次，也就意味着每天可能会发生多次集成。每次集成都通过自动化的构建（包括编译、发布、自动化测试）来验证，从而尽快地发现集成错误。经过众多的团队实践，得出持续集成的开发方式能减少软件集成问题，让团队专注软件本身逻辑。

持续集成的价值

提高软件质量

每天或每周进行多次的集成，并进行测试，有利于发现软件缺陷，最终提高软件质量。

减少重复劳动

持续集成通过自动化工具编译代码、打包程序、上传、部署、测试，无需太多人工干预，让开发拥有更多精力和时间专注于软件逻辑实现。

增强项目的预见性

持续集成过程中每次操作会记录详细输入输出结果，为后续分析产品缺陷是否收敛提供数据支撑。通过这些数据的分析增加了项目的预见性，可根据实际情况调整项目决策。

增强团队对产品的信心

持续集成可以增强团队对开发产品的信心，因为他们清楚知道软件做了哪些改动、可能会造成的影响，以及每一次构建的结果。

如何做到持续集成

1. 统一的代码库。
2. 自动构建工具。
3. 自动测试脚本。
4. 开发向代码库主干提交代码。
5. 自动触发构建工具。
6. 高速构建服务器，快速完成构建作业。
7. 上传软件到测试环境触发自动测试脚本。
8. 执行结果分析与展示。
9. 自动部署到演练环境。

持续集成管理工具

常见的持续集成管理工具有 Apache Continuum (<http://continuum.apache.org/>)、CruiseControl (<http://cruisecontrol.sourceforge.net/>)、Hudson (<http://hudson-ci.org/>)、Jenkins (<http://jenkinsci.org/>)、Luntnbuild (<http://luntnbuild.sf.net>) 等。这些工具对比可以参考 https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software。

Jenkins 是被广泛应用的持续集成、自动化测试、持续部署的框架，甚至可以当作流程管理的工具。当前 Jenkins 提供了 1000 多个插件用来支持编译、测试几乎所有的程

序。但 Jenkins 对于所运行的任务无法做到环境隔离，有了容器后可以很好地解决这一问题。

Jenkins 与 Kubernetes 结合实践

基于 Kubernetes 部署 Jenkins Master 比较简单，只需要在 Jenkins Master 安装 Kubernetes Plugin，并下载 Jenkins Slave 镜像。Jenkins 的任务通过 Kubernetes Plugin 转化为 Kubernetes Pod 模板，Kubernetes 根据 Pod 模板自动拉起 Slave 容器，Slave 容器会自动加入 Jenkins 集群。Jenkins 执行任务时自动调度 Job 到某个 Slave 容器，任务执行结束后，Jenkins 自动删除相关节点，并销毁对应容器，实现资源的释放。

Jenkins 与 Kubernetes 结合优点

整个底层资源分配和资源释放过程对用户来说是透明的，用户只需要创建好 Jenkins 任务，不需要关心节点在哪里；对于 Jenkins Master 来说，Slave 节点（容器）是临时的，任务一结束就会销毁。由于 Slave 运行在容器中，那么 Jenkins 调度的 Job 也会运行在容器中，每个容器是隔离的，即每个 Job 也实现了隔离。如果需要数据持久化可以在容器中挂载 NFS、GlusterFS、CephFS 等文件系统。

Jenkins 与 Kubernetes 方案解决了大型产品测试的效率和环境隔离问题，此方案在网易云基础服务多个项目中得到应用。

网易云基础服务持续集成 workflow

网易云基础服务持续集成是按各个业务功能模块分别持续交付，最终达到整个平台的持续集成。持续集成基本流程如图 4-43 所示。

1. 开发人员提交代码或脚本到源码仓库（SOURCE REPOSITORY）。
2. 编译服务器（CI SERVER）接收到源码编译任务，编译指定项目可执行文件（如 Go 语言项目同时编译和执行单元测试用例）。
3. 执行 Docker 命令打包可执行程序及相关环境变量、配置等为镜像。
4. 上传镜像到 DOCKER REGISTRY。
5. 拉取指定镜像版本执行全量单元测试用例。

6. 测试结果自动反馈给开发和测试。

7. 测试人员拉取镜像执行黑盒测试、可靠性测试等用例。执行结果直接反馈给开发和持续集成管理平台。持续交付是持续集成的下一步，做到持续集成后，很容易实现持续交付。

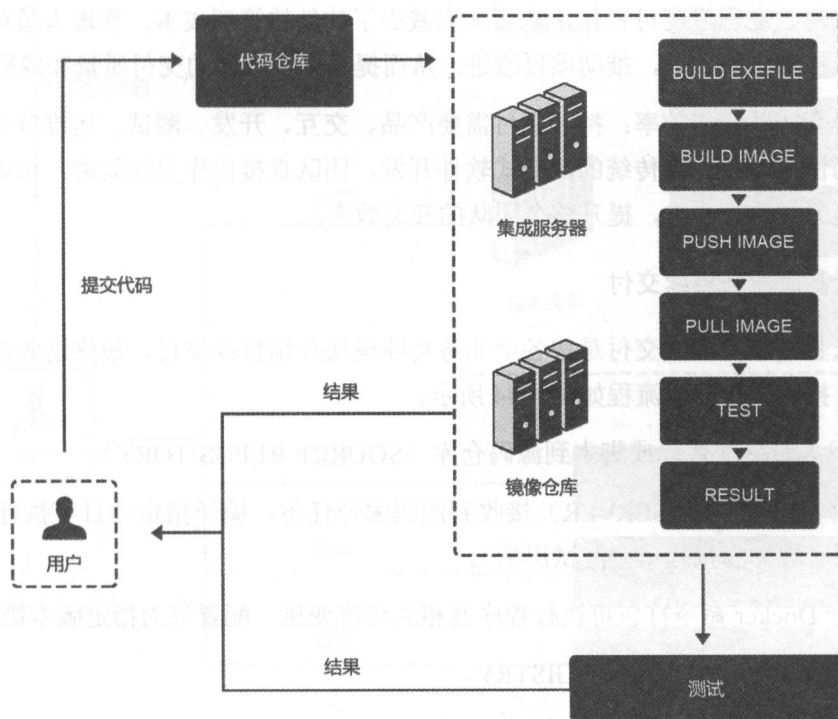


图 4-43 持续集成基本流程

4.6.2 持续交付

持续交付是指周期性地将新版本软件交付给质量运营团队或者用户，以供测试评审。如果评审通过，代码就进入生产阶段。持续交付的核心是不管有多少更新，软件始终可以交付。

持续交付的价值

持续交付的价值主要体现在以下 3 个方面。

- 快速发布产品新特性：持续交付可以缩短编码、测试、演练、上线、交付的迭代周期，使每个步骤中出现的问题得到快速响应。在产品上的体现就是缩短软件功能的发布周期，从而可以更好地应对业务需求变化。
- 高质量的软件发布标准：由于整个软件交付过程实现了标准化，整个交付过程可重复、过程进度可视化，从而大大减少了软件的管理成本，管理人员可以更容易地提高项目质量，推动项目改进，从而提高整个产品的交付质量和成熟度。
- 提高团队协作效率：持续交付需要产品、交互、开发、测试、运维等多个角色密切协作，相比于传统的瀑布式软件开发，团队直接协作更加紧密，可以减少团队之间的协调成本，提升整个团队的开发效率。

网易云基础服务持续交付

网易云基础服务持续交付是按各个业务功能模块分别持续交付，最终达到整个平台的持续交付。持续交付基本流程如图 4-44 所示。

1. 开发人员提交代码或脚本到源码仓库（SOURCE REPOSITORY）。
2. 编译服务器（CI SERVER）接收到源码编译任务，编译指定项目可执行文件（Go 语言项目同时编译和执行单元测试用例）。
3. 执行 Docker 命令打包可执行程序及相关环境变量、配置等为指定版本镜像。
4. 上传镜像到 DOCKER REGISTRY。
5. 拉取指定镜像版本执行全量单元测试用例。
6. 测试结果自动反馈给开发和测试。
7. 测试人员拉取镜像执行黑盒测试、可靠性测试等用例。执行结果直接反馈给开发。
8. 通过上面步骤测试和评估后进入演练环境（STAGING）试用或测试。执行结果直接反馈给开发。
9. 开发、测试、质量管理评估风险后进入产品试点环境。

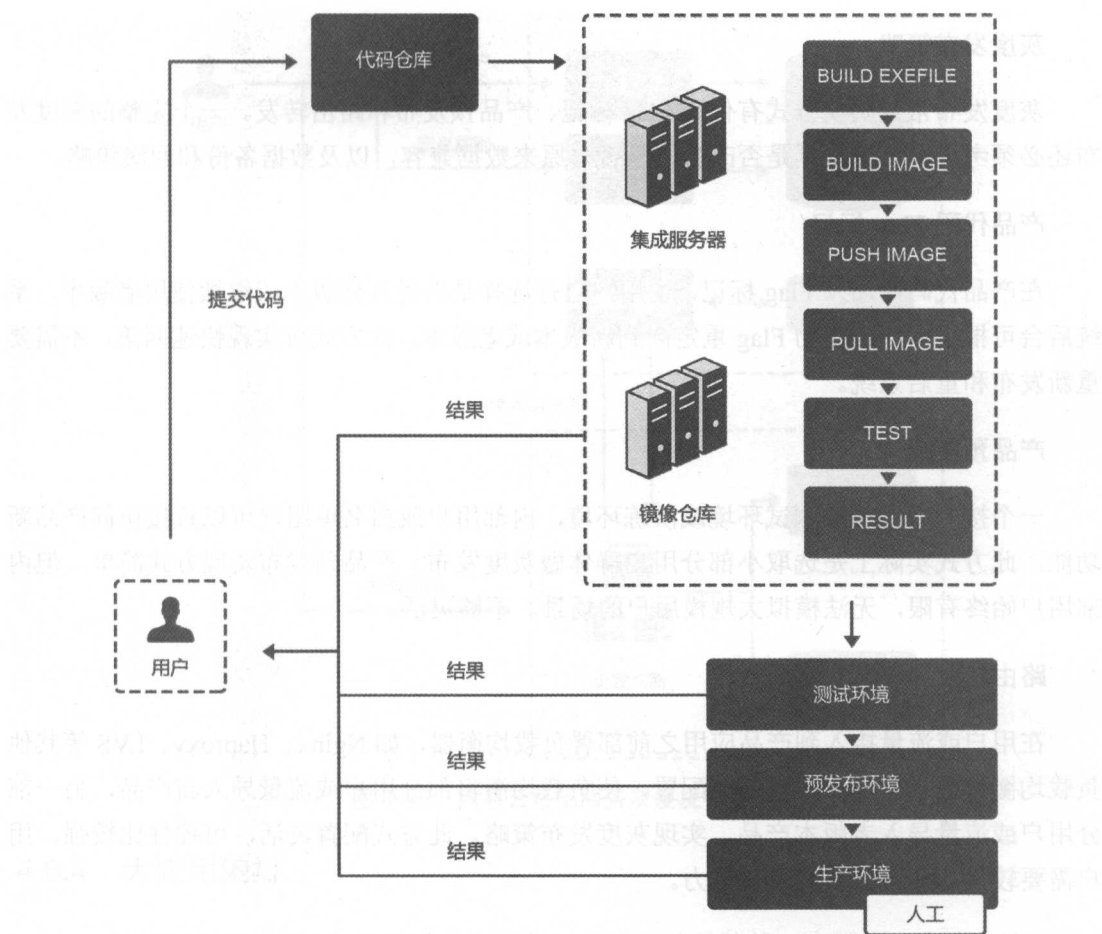


图 4-44 持续交付基本流程

4.6.3 灰度发布

灰度发布本质上是指选取部分用户或部分区域发布产品新特性，互联网产品的发布普遍采用灰度发布方式。A/B 测试是常用的灰度发布方式。A/B 测试按一定的策略选定一部分用户继续用 A，而另一部分用户开始用 B，产品可以通过使用 B 的用户反馈决策是否扩大发布范围，最终达到产品新特性向所有用户全量发布。灰度发布可以保证整体系统的可用性，在开始灰度的时候就可以发现、解决产品新特性带来的问题。

灰度发布策略

灰度发布常见实现方式有代码 **Flag** 标记、产品预发布和路由转发。一个完整的灰度发布还必须考虑有状态数据是否改变、是否与原来数据兼容，以及数据备份和回滚策略。

产品代码 **Flag** 标记

在产品代码中加入 **Flag** 标记，由用户自行选择是否进入新版本或继续使用老版本，系统后台可根据用户设置的 **Flag** 重定向到新版本或老版本。此方式可实现快速回滚，不需要重新发布和重启系统。

产品预发布

一个接近线运行的测试环境或演练环境，内部用户或白名单用户可以直接访问产品新功能。此方式实际上是选取小部分用户群体做灰度发布。产品预发布实现方式简单，但内部用户始终有限，无法模拟大规模用户的场景，不够灵活。

路由转发

在用户或流量接入到产品应用之前部署负载均衡器，如 Nginx、Haproxy、LVS 等其他负载均衡设备。采用一个灵活的配置，使负载均衡将部分用户或流量导入新产品，另一部分用户或流量导入老版本产品，实现灰度发布策略。此方式配置灵活，可控性比较强，用户需要较强的业务部署和管理能力。

回滚策略

不同的灰度发布实现方式有不同的回滚策略。

网易云基础服务灰度发布

如图 4-45 所示，网易云基础服务自身产品灰度发布主要使用 **Flag** 标记和路由转发的方式。用户可以自行选择是否体验新版本。选择体验新版本的用户信息中带有 **Flag** 标记，这个用户发起的所有请求都会被重定向到新版本试用环境中，其他用户请求访问正式环境。用户的请求也可以从新版本切回原来发布版本环境中。网易云基础服务为用户提供的灰度发布采用路由转发方式，用户产品需要用 **service** 的方式部署，灰度发布时可以通过 **Lable** 选择部分服务发布，用户还可以控制平台后端负载服务转化比率。

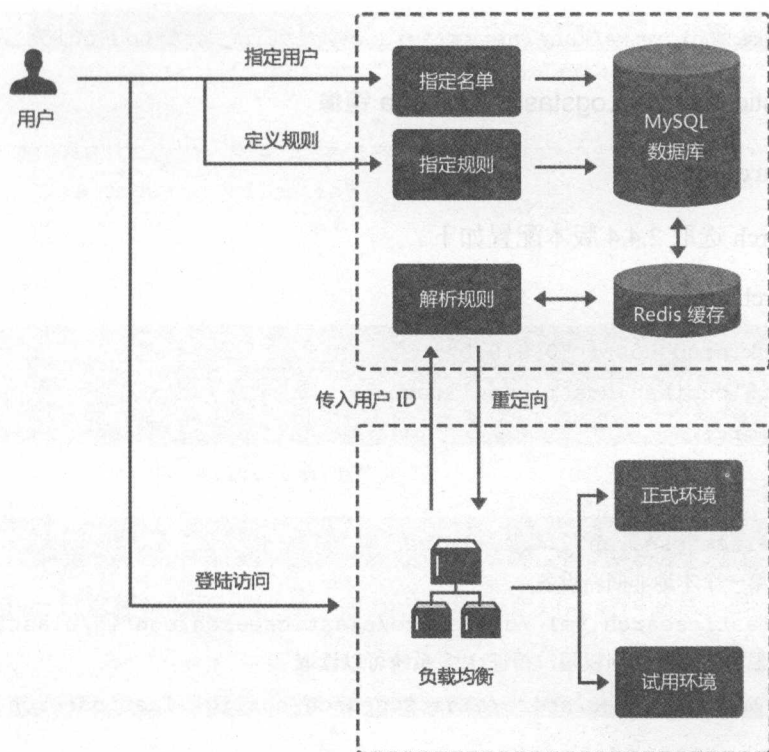


图 4-45 网易云基础服务灰度发布方案

4.6.4 大应用编排

一套大的应用方案包含多个模块和服务，通常做法是编排一套脚本，在脚本中定义好各个模块的配置和依赖，就可以用脚本在多个环境中部署。Docker 容器服务使应用编排更便捷，Docker Compose 的语法已在大多数云平台得到广泛的应用。

本节将介绍如何在原生的 Kubernetes 平台上编排一套 ELK (Elasticsearch、Logstash 和 Kibana) 应用服务。语法为 Yaml 格式。

原生 Kubernetes 平台搭建

搭建一个 Kubernetes 集群可以用开源软件 minikube 或 kubeadm。具体安装细节请参考官方网站。

minikube: <https://github.com/kubernetes/minikube>

```
kubeadm: https://kubernetes.io/docs/getting-started-guides/kubeadm/
```

构建 Elasticsearch、Logstash 和 Kibana 镜像

Elasticsearch

Elasticsearch 选取 2.4.4 版本配置如下。

elasticsearch.yml

```
network.bind_host: "0.0.0.0"
network.publish_host: _non_loopback_
http.port:
```

Dockerfile

```
FROM elasticsearch:2.4.4-alpine
# 监听第一个本地非回路设备
ADD elasticsearch.yml /usr/share/elasticsearch/config/elasticsearch.yml
# 修改配置文件的访问权限，得证 ES 系统可以读取
RUN chmod a+r /usr/share/elasticsearch/config/elasticsearch.yml
```

Logstash

Logstash 镜像选取 2.4.1 版本配置如下。

logstash.conf

```
input {
  courier {
    transport => "tcp"
    port => 8600
  }
}

filter {
  if [type] == "nginx" {
    grok {
      match => { "message" => "%{NGINXACCESS}" }
    }
  }
}
```



```

    date {
      match => [ "timestamp" , "dd/MMM/YYYY:HH:mm:ss Z" ]
    }
    geoip {
      source => "clientip"
    }
  }
}

output {
  elasticsearch {
    hosts => "elasticsearch"
  }
}

```

测试的 nginx 日志模板如下。

nginx

```

NGUSERNAME [a-zA-Z\.\@\-\+\_]+
NGUSER %{NGUSERNAME}
NGINXACCESS %{IPORHOST:http_host} %{IPORHOST:clientip}
\[ %{HTTPDATE:timestamp} \]
\"(?:%{WORD:verb} %{NOTSPACE:request} (? :
HTTP/%{NUMBER:httpversion})?| %{DATA:rawrequest})\" %{NUMBER:response}
(?:%{NUMBER:bytes}|-) %{QS:referrer} %{QS:agent}
%{NUMBER:request_time:float} %{NUMBER:upstream_time:float}
NGINXACCESS %{IPORHOST:http_host} %{IPORHOST:clientip}
\[ %{HTTPDATE:timestamp} \] \"(?:%{WORD:verb} %{NOTSPACE:request} (? :
HTTP/%{NUMBER:httpversion})?| %{DATA:rawrequest})\"
%{NUMBER:response} (?:%{NUMBER:bytes}|-)
%{QS:referrer} %{QS:agent} %{NUMBER:request_time:float}

```

本次使用的日志 agent 为 log courier，所以需要安装 logstash-input-courier 插件。

Dockerfile

```
FROM logstash:2-alpine
```

```

MAINTAINER Vincent Ambo <vincent@nordcloud.com>
WORKDIR /opt/logstash
EXPOSE 8600
# 在 nginx 配置中添加格式过滤器
ADD nginx /opt/logstash/patterns/nginx
# 安装 log-courier 输入插件，下面地址是对应的说明文档
#
https://github.com/driskell/log-courier/blob/master/docs/LogstashIntegration.md
RUN bin/plugin install logstash-input-courier
# 添加 logstash 配置文件
ADD logstash.conf /etc/logstash.conf
# 复写对应命令
CMD ["logstash", "agent", "-f", "/etc/logstash.conf"]

```

Kibana

Kibana 的配置如下。

kibana.yml

```

# 配置 Kibana 服务器的监听端口
server.port: 5601
# 绑定的 IP 地址，"0.0.0.0"表示在所有地址上监听
server.host: "0.0.0.0"
# Elasticsearch 实例对应的地址
elasticsearch.url: "http://elasticsearch:9200"

```

Dockerfile

```

FROM alpine
# 设置环境变量
ENV KIBANA_VERSION 4.6.4
ENV PKG_NAME kibana
ENV PKG_PLATFORM linux-x86_64
ENV KIBANA_PKG $PKG_NAME-$KIBANA_VERSION-$PKG_PLATFORM
ENV KIBANA_CONFIG /opt/$PKG_NAME-$KIBANA_VERSION-$PKG_PLATFORM/config/
kibana.yml ENV KIBANA_URL https://download.elastic.co/\$PKG\_NAME/\$PKG\_NAME/

```

```

$KIBANA_PKG.tar.gz
ENV ELASTICSEARCH_HOST elasticsearch
RUN addgroup -S kibana && adduser -S -G kibana kibana
# 下载 Kibana
RUN apk add --update ca-certificates wget nodejs \
    && mkdir -p /opt \
    && wget -O /tmp/$KIBANA_PKG.tar.gz $KIBANA_URL \
    && tar -xvzf /tmp/$KIBANA_PKG.tar.gz -C /opt/ \
    && ln -s /opt/$KIBANA_PKG /opt/$PKG_NAME \
    && sed -i "s/localhost/$ELASTICSEARCH_HOST/" $KIBANA_CONFIG \
    && rm -rf /tmp/*.tar.gz /var/cache/apk/* /opt/$KIBANA_PKG/node/ \
    && mkdir -p /opt/$KIBANA_PKG/node/bin/ \
    && ln -s $(which node) /opt/$PKG_NAME/node/bin/node \
    && chown -R kibana:kibana /opt
# 暴露服务
EXPOSE 5601
# 添加 Kibana 默认配置
ADD kibana.yml /opt/kibana/config/kibana.yml
USER kibana
# 设置工作目录
WORKDIR ["/opt/kibana"]
CMD ["/opt/kibana/bin/kibana"]

```

Kubernetes 原生编排模板

本测试为实验性质故将 ELK 3 个镜像配置在同一个 pod 中, 然后暴露 logstash 和 kibana 两个服务, 应用直接访问 logstash 和 kibana 服务。配置模板如下。

elk-deploy.yml

```

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: elk-rc
  labels:
    k8s-app: elk

```

```
spec:
  template:
    metadata:
      labels:
        k8s-app: elk
    spec:
      containers:
        - image: hub.c.163.com/gobyong/elasticsearch:2.4.0-elk
          name: elasticsearch
          ports:
            - containerPort: 9200
          volumeMounts:
            - name: es-storage
              mountPath: /usr/share/elasticsearch/data
        - image: hub.c.163.com/gobyong/logstash:2-elk
          name: logstash
          ports:
            - containerPort: 8600
        - image: hub.c.163.com/gobyong/kibana:4-elk
          name: kibana
          env:
            - name: ELASTICSEARCH_URL
              value: http://127.0.0.1:9200
          ports:
            - containerPort: 5601
      volumes:
        - name: es-storage
          emptyDir: {} # 由于 es 是有状态的，这里可以改成有状态盘
```

elk-svc-kibana.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kibana
  labels:
```



```
k8s-app: elk
spec:
  selector:
    k8s-app: elk
  ports:
    - port: 5601
    name: kibana
```

elk-svc-logstash.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: logstash
  labels:
    k8s-app: elk
spec:
  selector:
    k8s-app: elk
  ports:
    - port: 8600
    name: logstash
```

创建 pod 和服务

```
kubectl create -f elk-deploy.yaml
kubectl create -f elk-svc-logstash.yaml
kubectl create -f elk-svc-kibana.yaml
kubectl get pods -o wide
kubectl get services -o wide
```

最后打开 <http://kibana.default.svc.cluster.local:5601> 即可调用服务。

测试镜像准备

使用 nginx 作为服务器进行测试，对应的 nginx.conf 文件内容如下。

```
worker_processes 1;

error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;

    log_format logstash '$http_host ' '$remote_addr [$time_local] '
        '"$request" $status $body_bytes_sent '
        '"$http_referer" "$http_user_agent" ' '$request_time '
        '$upstream_response_time';
    access_log /var/log/nginx/access.log logstash;

    server {
        listen 80;
        server_name localhost;

        location / {
            root /usr/share/nginx/html/k8s-elk-demo;
            index index.html;
        }
    }
}
```

Dockerfile

```
FROM nginx:alpine
ADD nginx.conf /etc/nginx/nginx.conf
```

数据收集端为 log-courier，配置如下。

log-courier.conf

```
{
  "network": {
    "servers": [ "logstash:8600" ],
    "transport": "tcp"
  },
  "files": [
    {
      "paths": [ "/var/log/nginx/access.log" ],
      "fields": { "type": "nginx" }
    }
  ]
}
```

Dockerfile

```
FROM centos:7
# 安装 log-courier
ADD https://copr.fedoraproject.org/coprs/driskell/log-courier/repo/epel-7/driskell-log-courier-epel-7.repo /etc/yum.repo s.d/
RUN yum install -y epel-release --nogpgcheck && yum install -y log-courier --nogpgcheck && yum install nginx
# 配置 log-courier
ADD log-courier.conf /etc/log-courier/log-courier.conf
CMD log-courier -config=/etc/log-courier/log-courier.conf
```

测试 Deployment

要进行集成测试，需要 log-courier 和 nginx 容器共享日志、存储盘 log-storage（测试时不考虑两个容器共享目录的写一致问题）。

demo.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  name: demo-service
  labels:
    k8s-app: demo-app
spec:
  type: LoadBalancer
  selector:
    lb-target: web
  ports:
    - port: 80
      name: http
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: demo-app-v1
  labels:
    k8s-app: demo-app-v1
spec:
  replicas: 3
  template:
    metadata:
      labels:
        k8s-app: demo-app-v1
        lb-target: web
    spec:
      containers:
        - image: hub.c.163.com/gobyong/elk:log-courier-latest
          name: log-courier
          volumeMounts:
```

```

      - name: log-storage
        mountPath: /var/log/nginx
- image: hub.c.163.com/gobyong/elk:nginx-demo
  name: nginx
  ports:
    - containerPort: 80
  volumeMounts:
    - name: log-storage
      mountPath: /var/log/nginx
    - name: www-storage
      mountPath: /usr/share/nginx/html
  volumes:
    - name: www-storage
  gitRepo:
    repository: https://github.com/tazjin/k8s-elk-demo.git
    revision: static-v1
    - name: log-storage
  emptyDir: {}

```

生成模板

根上面的测试最终生成网易云基础服务平台下的一个 elk 编排模板为 elk.yaml。

```

---
# Kibana 服务
apiVersion: v1
kind: Service
metadata:
  name: kibana
  namespace: laidonglin-k8sk8sk8s # change to your ns
  labels:
    name: elk-demo-kibana
spec:
  selector:
    name: elk-demo-kibana
  ports:

```



```
- port: 5601
  name: kibana
---
# LogStash 服务
apiVersion: v1
kind: Service
metadata:
  name: logstash
  namespace: laidonglin-k8sk8sk8s
  labels:
    name: elk-demo-logstash
spec:
  selector:
    name: elk-demo-logstash
  ports:
    - port: 8600
      name: logstash
---
# Elasticsearch 服务
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
  namespace: laidonglin-k8sk8sk8s
  labels:
    name: elk-demo-es
spec:
  selector:
    name: elk-demo-es
  ports:
    - port: 9200
      name: elasticsearch
---
apiVersion: extensions/v1beta1
kind: Deployment
```



```
metadata:
  name: elk-demo-es
  namespace: laidonglin-k8sk8sk8s
  labels:
    name: elk-demo-es
spec:
  replicas: 1
  minReadySeconds: 10
  revisionHistoryLimit: 1
  template:
    metadata:
      labels:
        name: elk-demo-es
        name: elk-demo-es
        namespace: laidonglin-k8sk8sk8s
    spec:
      containers:
        - image: hub.c.163.com/elkdemo/elasticsearch:2.4.4-alpine
          name: elasticsearch
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 9200
          volumeMounts:
            - name: es-storage
              mountPath: /usr/share/elasticsearch/data
          resources:
            limits:
              cpu: "1"
              memory: "1073741824"
            requests:
              cpu: "1"
              memory: "1073741824"
      volumes:
        - name: es-storage
          emptyDir: {}
```

```
# 为部署设置网络和节点选择器
node:
  cpu: 1000m
  memory: "1073741824"
networks:
  - netType: lan
nodeSelector:
  stateful: "false"
  tenantid: 6089d765c34a446e93778e1cd4133f72
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: elk-demo-logstash
  namespace: laidonglin-k8sk8sk8s
  labels:
    name: elk-demo-logstash
spec:
  replicas: 1
  minReadySeconds: 10
  revisionHistoryLimit: 1
  template:
    metadata:
    labels:
      name: elk-demo-logstash
      name: elk-demo-logstash
      namespace: laidonglin-k8sk8sk8s
    spec:
      containers:
        - image: hub.c.163.com/elkdemo/logstash:2-alpine
          name: logstash
          ports:
            - containerPort: 8600
          resources:
            limits:
```

```

      cpu: "1"
      memory: "1073741824"
    requests:
      cpu: "1"
      memory: "1073741824"
  node:
    cpu: 1000m
    memory: "1073741824"
  networks:
  - netType: lan
  nodeSelector:
    flavorid: "176"
    stateful: "false"
    tenantid: 6089d765c34a446e93778e1cd4133f72 # 必须的
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: elk-demo-kibana
  namespace: laidonglin-k8sk8sk8s
  labels:
    name: elk-demo-kibana
spec:
  replicas: 1
  minReadySeconds: 10
  revisionHistoryLimit: 1
  template:
    metadata:
      labels:
        name: elk-demo-kibana
    namespace: laidonglin-k8sk8sk8s
  spec:
    containers:
      - image: hub.c.163.com/elkdemo/kibana:4.6.4-alpine

```

```
name: kibana
env:
  - name: ELASTICSEARCH_URL
    value: http://elasticsearch:9200
ports:
  - containerPort: 5601
resources:
  limits:
    cpu: "1"
    memory: "1073741824"
  requests:
    cpu: "1"
    memory: "1073741824"
#为部署设置网络和节点选择器
node:
  cpu: 1000m
  memory: "1073741824"
networks:
  - netType: lan
nodeSelector:
  stateful: "false"
```

4.7 安全设计

在云计算的背景下，大量的业务从传统基础架构迁移到了云上，但是仍然面对传统的网络安全威胁。目前，网络中蠕虫、病毒、木马仍然威胁巨大，各种网络攻击行为时有发生，随着 APT 攻击的流行，企事业单位正遭受着前所未有的安全威胁。云计算的引入，又带来了虚拟化安全、数据安全、安全法规等新的安全挑战。

随着网络的普及，攻击工具朝着自动化、易用化、平民化的方向发展，攻击者无需掌握太多的专业知识，只需要从网上下载一个自动化工具鼠标一点就可以完成一次攻击，导致攻击的成本和门槛越来越低。是否能够及时发现网络黑客的入侵行为，保证系统安全，成为所有网络用户包括云计算用户所面临的一个重要问题。

4.7.1 入侵检测

针对日趋复杂的应用安全威胁和混合型网络攻击，各类企事业单位为了保护好自身的应用做了很多的工作，各种乙方安全公司也提供一系列的安全产品和解决方案。各类新的服务层出不穷，例如 Web 应用、H5 应用、App 应用，在给用户带来更好体验的同时，也带来了前所未有的复杂性和危险性。入侵检测系统 IDS 是检测和发现网络攻击的一种常规手段，根据部署的位置不同，大致可以分为基于网络入侵检测系统 NIDS 和基于主机的入侵检测系统 HIDS 两种。

- 基于网络的入侵检测 NIDS 是传统入侵检测系统，主要通过旁路在计算机网络中检测设备和系统，对网络数据流量进行深度检测和分析，并对网络中的攻击行为和特征进行主动检测匹配，如图 4-46 所示。通过使用异常检测、协议分析、会话分析、实时关联检测、机器学习等多种技术手段，系统可以实现对网络流量的入侵检测。

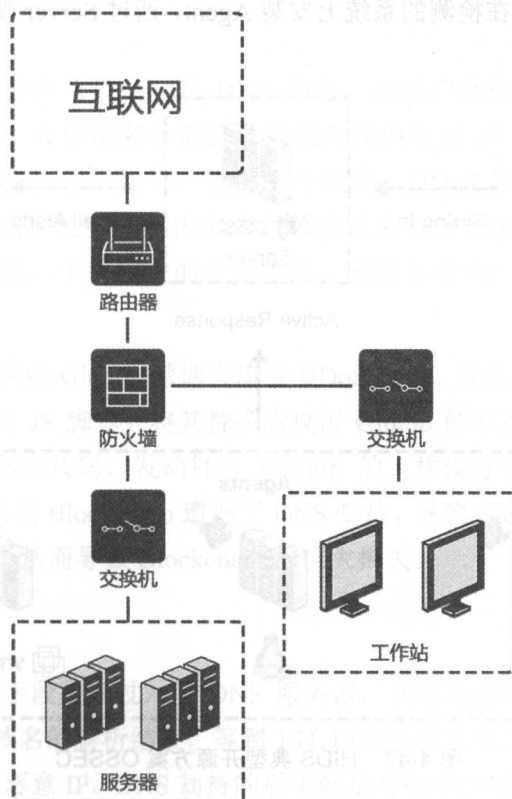


图 4-46 传统入侵检测系统

经典的开源解决方案有 Snort。Snort 通过对网络的数据包进行嗅探和实时分析，进行规则的匹配和处理之后，输出各类报警、忽略、Log 等多种处理方式。

- 基于主机的入侵检测系统 HIDS 是云计算中的入侵检测系统，通常通过安装在主机上 Agent 程序对该主机的网络实时连接、文件系统、日志进行智能分析和判断。当有文件发生变化时，HIDS 将文件与攻击特征样本库进行对比，看它们是否匹配。如果匹配，HIDS 就会向相关人员报警，相关人员在确认之后采取对应的措施。基于主机的 IDS 在发展过程中融入了其他技术。HIDS 还可以监控端口和进程来发现入侵行为。对于常用服务端口的暴力破解和异常登录也是 HIDS 常见的功能，能在很大程度上缓解内外部的攻击行为。

HIDS 典型的开源方案是 OSSEC。如图 4-47 所示，OSSEC 支持多种操作系统，如 Windows、Linux、MacOS 等，它包括了日志分析、rootkit 检测、文件异常检测等功能。OSSEC 使用 CS 方式部署，需要在检测的系统上安装 Agent，通过 Server 端预设的规则进行分析和报警处理。

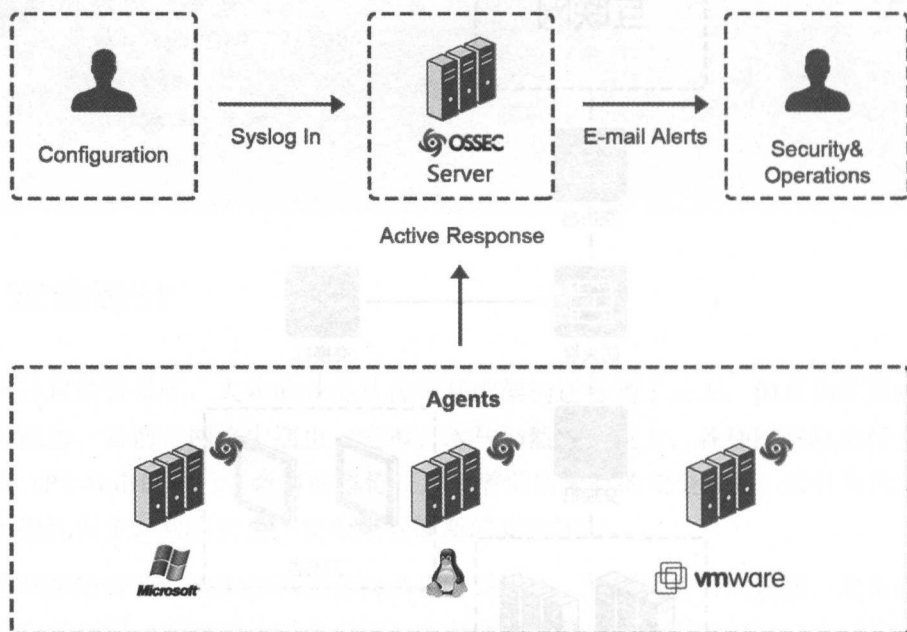


图 4-47 HIDS 典型开源方案 OSSEC

入侵检测实践

目前云计算常见的入侵检测方案也分为网络和主机两种，网络检测方案更多用于特定的应用场景下，对用户提供更多的是基于主机的入侵检测功能，需要用户配合在虚拟机或者容器中安装相应的 Agent 客户端进行监控。

一般的入侵检测 Agent 包含检测、修复、防御等功能，提供全面的木马查杀、0day 漏洞修复、安全基线巡检、主机访问控制、暴力破解防护、异地登录提醒等功能，保障服务器安全。

4.7.2 防劫持攻击

我们在访问某些特定站点或者是所有站点时，有时会弹出广告。比较典型的是，当宽带快要欠费的时候，运营商会弹出页面提醒你续费。这是怎么做到的呢？其实就是通过流量劫持的方式。

流量劫持方式主要分为两类：一是 DNS 劫持，在用户通过域名系统访问目标的时候劫持域名解析的回包，目标域名会被恶意地指向其他恶意 IP 地址，造成用户无法正常使用服务，甚至造成弹广告、挂马、搜集账号密码、DDoS 等更加严重的危害；二是链路劫持，对于服务器返回的数据包内容进行篡改或者抢先与服务器进行回包，在回包中强行插入恶意业务逻辑，干扰用户的正常使用，同样会造成广告、挂马等一系列的危害后果。

2015 年，代码托管网站 GitHub 遭遇大流量 DDoS 攻击。攻击者使用了链路劫持技术，通过劫持百度广告联盟的 JS 脚本并将其替换成攻击 Github 的恶意代码，然后访问百度海量用户的浏览器加载改恶意代码，发动针对 GitHub 的大规模分布式拒绝服务攻击。2016 年，美国最大的比特币公司 Blockchain 遭遇了 DNS 劫持，导致 Blockchain 的用户正常登录后被引导至错误的页面，从而导致 Blockchain 的巨大损失。

DNS 劫持

DNS 劫持最直接的手段是通过入侵 DNS 服务器，获取攻击目标域名的解析记录控制权，之后修改攻击目标域名的解析结果，等到 TTL 时间生效之后，所有对该域名的访问由原 IP 地址指向修改后的恶意 IP。DNS 劫持的后果就是对被修改域名记录的网址不能访问，甚至访问到虚假网址，导致用户数据被盗、被挂马的严重危害。

通常的 DNS 查询没有强大的认证机制作为安全保证，而且 DNS 查询通常基于无连接不可靠的 UDP 协议，因此 DNS 的查询非常容易被篡改，通过对 UDP 端口 53 上的 DNS 查询进行侦听，一经发现与关键词相匹配的请求，则立即伪装成目标域名的解析服务器（NS，Name Server）给查询者返回虚假结果，从而实现窃取资料或者破坏原有正常服务的目的。还有一种手段是黑客伪造权威 DNS 服务器的应答，对缓存 DNS 服务器进行污染。

DNS 劫持对策

DNS 劫持往往是针对个人，分布在不同的运营商和不同的网络中。就个人而言，针对 DNS 劫持，可以采用使用国外免费公用的 DNS 服务器解决，例如 GoogleDNS（8.8.8.8）；针对 DNS 污染，可以使用 VPN 或者域名远程解析的方法解决，但这大多需要购买付费的 VPN 或 SSH 等，还可以通过修改本地 Host 文件的方法，手动设置域名正确的 IP 地址，但这样会大大提高维护成本。

对于服务提供方，需要对 DNS 劫持进行持续的检测。大量的终端布点可以检测 DNS 劫持的情况，帮用户针对劫持向运营商投诉。同时，现在越来越多的互联网厂商开始使用 HTTPDNS 服务，能够从一定程度上缓解 DNS 劫持的影响。HTTPDNS 基本原理如图 4-48 所示。

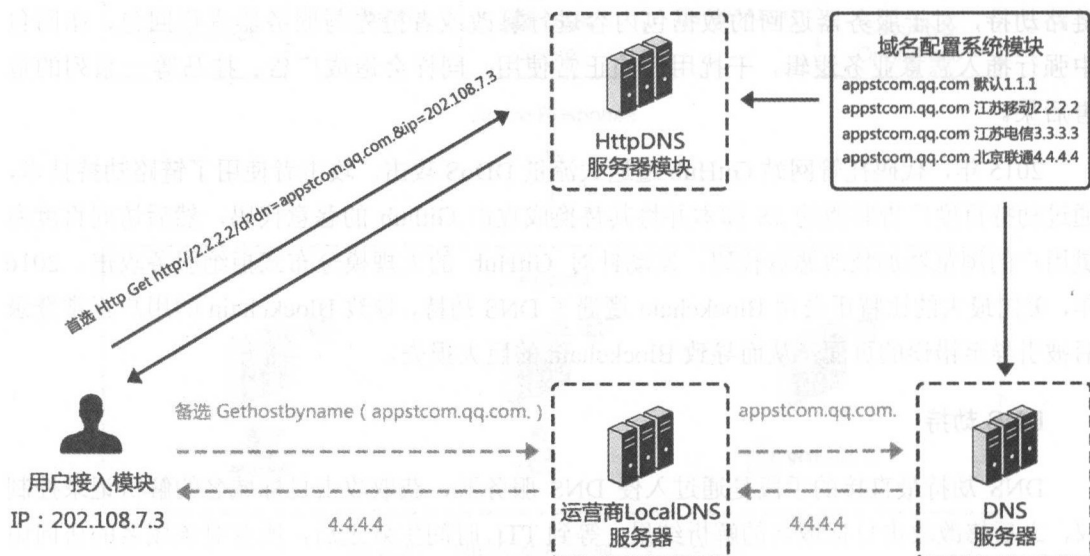


图 4-48 HTTPDNS 基本原理

如何使用 HTTPDNS 服务

开通服务，在 HTTPDNS 服务提供商处注册账号，开通 HTTPDNS 服务。

控制台添加域名

在 HTTPDNS 提供商的服务台申请 HTTPDNS 服务，这可能涉及费用（一般的 HTTPDNS 服务提供商会有一定的免费查询次数，如果超过了这个次数就需要额外的费用）。假设我们通过控制台获得了 HTTPDNS 服务商提供的账号，就直接在控制台上提价所需要解析的域名记录，一般几分钟之后会在 HTTPDNS 服务端生效。

使用 HTTPDNS 解析域名

使用 HTTPDNS 解析域名，可以使用类似下面的方式进行。`http://httpdns-service.com?host=www.yourdomain.com`，其中 `http://httpdns-service.com` 是 HTTPDNS 的提供方，`www.yourdomain.com` 是需要解析的域名。具体的 API 接口规范可参考 HTTPDNS 的服务提供商。

客户端集成 HTTPDNS

在通过 HTTPDNS 设置获得域名的 IP 地址后，客户端应用能够利用这个 IP 发起业务请求。

链路劫持

链路劫持是指第三方（可能是运营商、黑客或者内鬼）通过在用户客户端至服务器之间的网络线路中植入恶意设备，侦听用户和服务器之间的通信数据，达到窃取和篡改用户重要数据的目的。链路劫持最常用的就是在网页中植入广告，当然也可以进行挂马、钓鱼，还可以窃取帐号密码、银行卡、身份证等重要个人数据。

链路劫持的原理是侦听网络上的数据包，发现某些设定特征的包时，抢在服务器回包之前进行回包，这样客户端会自动忽略服务器真正的响应包而接受先到的假回包，从而受到欺骗和劫持。

链路劫持对策

从链路劫持的原理来看，其实是网络链路上侦听、伪造 TCP 包，达到控制目标网络链路的行为。我们发现核心的问题是，客户端没有办法识别返回的应答是服务器返回的，还

是第三方返回的，所以信任了第三方的应答，丢弃了真正的应答。解决这个问题可以通过全程 HTTPS 来解决，如图 4-49 所示，HTTPS 使用了加密技术，第三方无法知道通信的密钥，保证通信的安全而不被劫持。

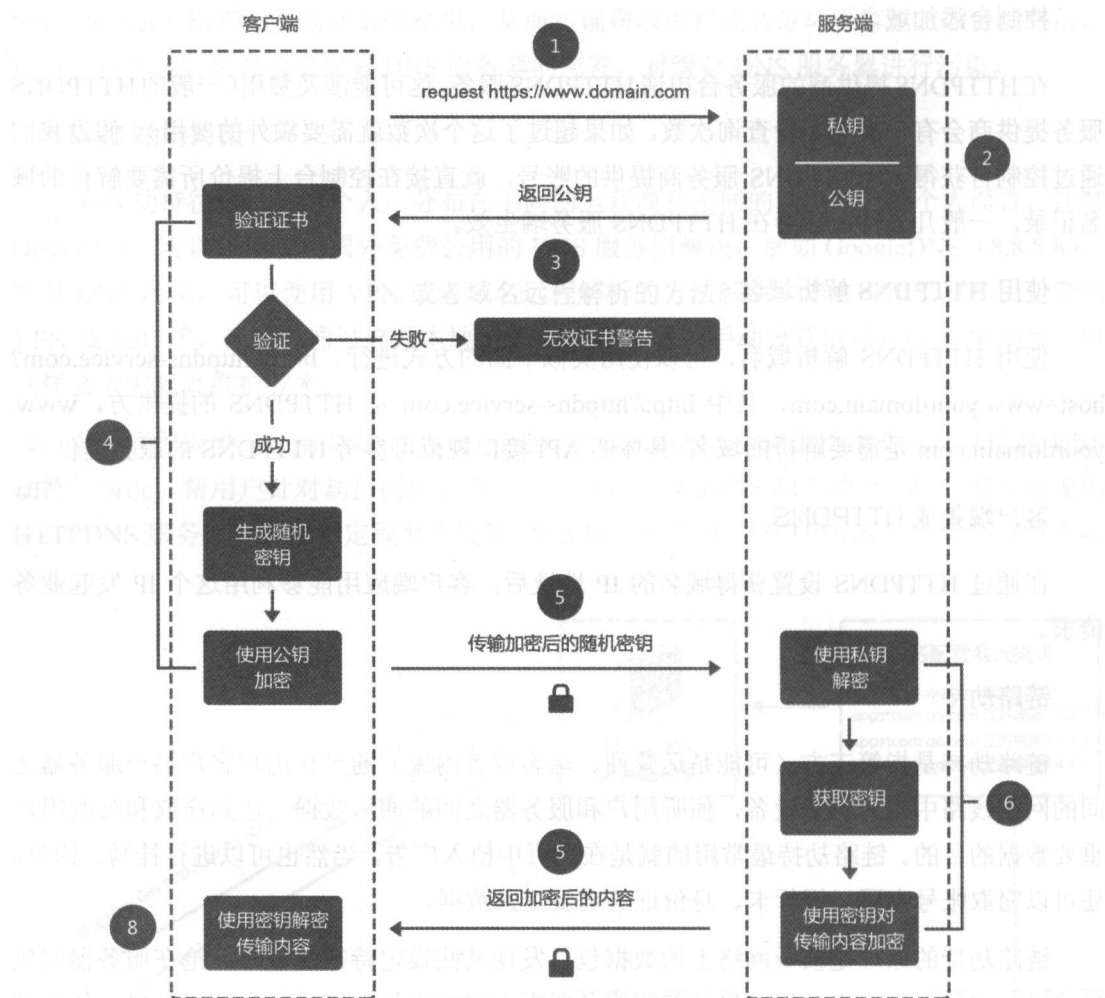


图 4-49 通过全程 HTTPS 解决链路劫持问题

HTTPS 服务部署实践

获取 CA 证书

SSL 服务首先需要申请 SSL 证书。证书可以分成付费和免费的两种，产品可以根据自身

的需求进行选择。在申请证书的时候，需要提供大量的信息以供 CA 机构备案和签署证书。

以网易云基础服务为例，通过平台申请 CA 证书需要点击左侧导航栏“SSL 证书管理”，进入 SSL 证书管理页面，再点击“申请证书”按钮开始申请，经过“确认证书信息→确认公司信息→确认联系人信息→绑定域名”等步骤之后，点击“提交申请”按钮验证域名所有权，弹出如图 4-50 所示的对话框。

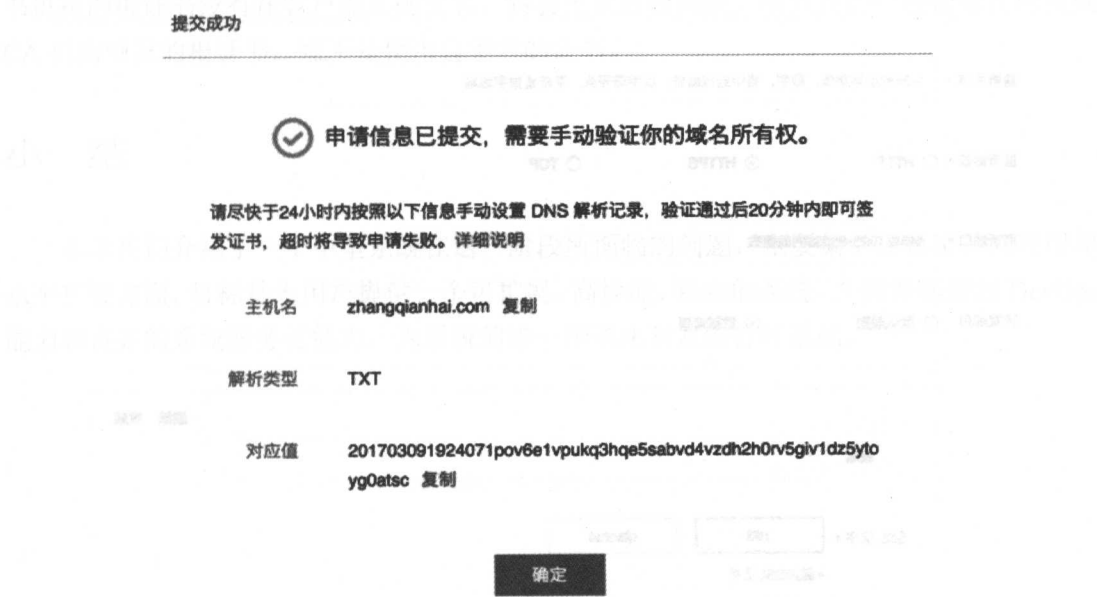


图 4-50 验证域名所有权对话框

申请信息成功提交后，可以在 SSL 证书管理页面看到证书状态为“验证中”，在该页面也可以查看域名所有权的验证方式，如图 4-51 所示。更详细的步骤，请参阅官方使用文档。

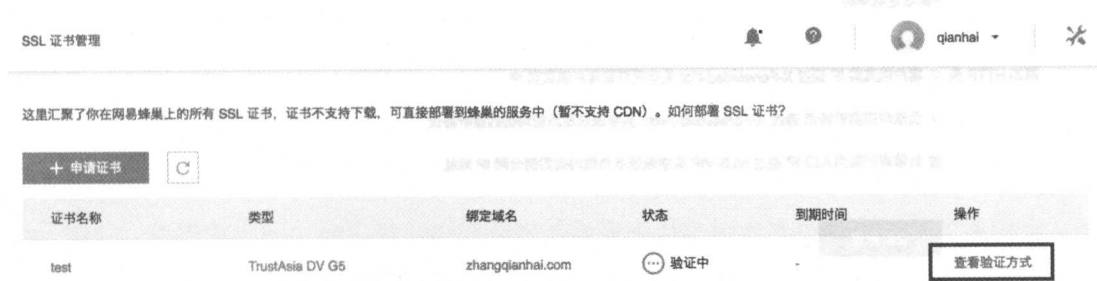


图 4-51 证书状态为“验证中”

部署 CA 证书

现在一般的网站会使用负载均衡，所以 CA 证书需要部署到负载均衡实例上。以网易云基础服务为例，可以在创建监听时选择“HTTPS”监听协议，然后在 SSL 证书处选择已经申请的证书，如图 4-52 所示。

< 创建监听

监听名称 *

1-24位小写字母、数字、或中划线组成，以字母开头，字母或数字结尾

监听协议 *

☐ HTTP ☒ HTTPS ☐ TCP

监听端口 *

443或1025-65535内的整数

转发规则

☐ 默认规则 ☒ 定制规则

删除 收起

域名

SSL 证书 *

163 qianhai

+添加SSL证书

URL 配置 * URL 服务 服务端口 健康检查规则 会话保持

无服务 ▾

1-65535P

默认规则 修改

☐ 启用

+

+新增定制规则

附加 HTTP 头

☒ 客户端真实 IP 通过 X-Forwarded-For 头字段获取客户端真实 IP

☒ 负载均衡监听协议 通过 X-Forwarded-Proto 头字段获取负载均衡的监听协议

☒ 负载均衡实例入口 IP 通过 NLB-VIP 头字段获取负载均衡实例公网 IP 地址

立即创建

图 4-52 部署 CA 证书

254

如果没有使用负载均衡设备,就需要在后端的 Web 服务器中配置 SSL 证书,具体可以参考各类不同的 Web 服务器配置文档进行配置。

客户端安装 CA 证书

客户端一般已经内置权威 CA 机构的根证书,不需要额外配置安装证书。如果颁发证书机构的根证书没有在客户端正确安装,将会出现连接问题。所以我们一般推荐使用权威 CA 机构颁发的根证书,而不是使用自签名的证书。

小 结

本章我们介绍了一个中型系统在这一阶段所面临的问题,主要集中在稳定、高可用和水平扩展方面,目标是为用户提供一个可扩展、高性能、稳定的系统,并提供基础的 DevOps 能力和良好的系统服务化能力,为系统的进一步进化和发展打好基础。

第5章 稳定期服务化应用架构实践

一个稳定日活百万流量的系统，通常的 MVC 垂直单体架构已经无法满足这一阶段的业务需求，业务复杂性的增加会降低团队开发效率，同时复杂的单体架构应用对产品的用户体验也会存在影响，这个时期的一个关键需求是要进行服务拆分，把一个服务拆成多个服务，各个服务分而自治、各司其职、协同工作，共同完成业务逻辑。随着服务拆分，配置项和配置文件会分散化，管理不慎就会变得非常混乱，难以维护，这时引入统一配置中心能有效解决这个问题。另外，分布式定时任务系统和分布式锁系统也是这个阶段的共性需求，比如对账系统对分布式定时任务系统的依赖，以及协调服务对分布式锁系统的依赖等。

服务化架构的出现，是为了解决软件研发成本高、测试部署维护代价大、扩展性差等问题。从 1996 年到现在，服务化的进程主要经历了两个阶段：伴随着 SOA（面向服务架构）概念的出现，以及近两年红火的微服务架构。服务化的核心在于如何做好服务发现、服务治理，以及解决服务化之后带来的数据一致性和全链路跟踪问题。另外，服务化也为高可用带来了新的机遇和挑战，一方面，业务可以通过实施服务治理和同城多活架构提升可用性，另一方面，同城多活背后的技术复杂性令人望而止步，多数人可能会采用云原生应用架构，选择基于云平台来快速获得业务同城多活能力。本章主要介绍成熟稳定期的应用挑战和架构方法。

5.1 业务拆分

在对业务进行拆分之前，我们先来了解一下整体的拆分原则。

拆分原则

在软件工程里，高内聚性和低耦合性是衡量分层设计好坏的一个重要指标。同样，在分布式服务化架构中，高内聚和低耦合也可以作为业务拆分的指导原则，本质上，这是一种单一职责和职责分离的体现，满足高内聚和低耦合的服务更易开发和运维。

高内聚

内聚性衡量服务内部互相联系的紧密程度。服务内部各模块联系越紧密，内聚性越高，反之，内聚性越低。在一个低内聚的分布式服务系统里，可能需要同时修改多个服务来满足功能交付，这样不仅引入开发工作量和复杂性，还扩大了服务发布带来的风险，应该尽量避免。解决办法就是设计功能高度内聚的服务，使得每次修改只涉及较少服务，以达到快速发布和交付的目的。

低耦合

耦合性衡量服务间互相联系的紧密程度。服务间联系越紧密，耦合性越高，反之，耦合性越低。如果服务间耦合过紧，就会出现对一个服务的修改导致另一个服务被动也需要修改的问题。低耦合的架构也有助于软件功能快速交付。

要实现服务化系统架构的高内聚和低耦合，必须首先定义清楚服务边界。这里的“边界”可以是业务边界，也可以是技术边界。在此主要讨论业务边界的界定。

如何进行业务拆分

当我们讨论业务拆分的实施时，不同的阶段、不同的上下文面临的挑战也不尽相同。

对于一个全新设计的系统，我们的关注点在于业务边界确定、服务间通信方式的选择等问题上。业务边界的确定过程，即服务建模的过程，这是服务拆分中最重要的工作。有些边界是很好定义的，比如用户服务和商品展示服务是两个几乎完全独立的业务，很少有共享的数据模型。但是另外一些边界就不是那么直观了，比如支付服务和结算服务会共享包括账户余额在内的多类数据。此时就需要对业务进一步思考，到底是边界不直观，还是这两个业务本身就应该属于同一服务，对于这种场景，我们的建议是先将这两块业务设计为同一个服务，等业务边界能够被稳定定义出来之后再做拆分，过早定义边界不明的服务会导致很多跨服务的修改，甚至引入循环依赖，违背了松耦合原则。当一个系统的业务边界都明确之后，我们可以采用以下步骤拆分。

- 数据库独立。不同的业务使用不同的数据访问层访问各自数据库。
- 代码独立。特别是采用 MVC 模式的垂直应用架构，需根据业务特征抽离出不同的实现类，使用独立代码仓库进行管理。

- 确定服务间通信方式，连接服务。服务间的通信有同步、异步或者同步异步相结合 3 种方式，这里的同步和异步是指是否需要服务双方同时在线。同步方式的选择有 RESTful (HTTP Based)、分布式服务框架等。异步方式可以使用消息队列服务器等。此处不再详述服务间通信方式的选择，感兴趣的读者可以参考第 4 章的相关章节。
- 服务独立发布与部署。

对于一个历史遗留系统的服务化改造，我们的关注点着重于系统架构的平滑过渡。如何不中断业务，实现“为一架飞行中的飞机更换引擎”是一个很大的技术挑战。我们建议采用的策略是“逐步改造、平滑过渡”。具体步骤如图 5-1 所示。

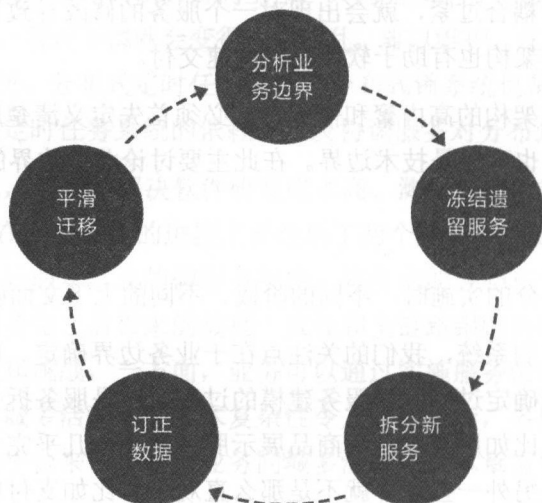


图 5-1 历史遗留系统的服务化改造流程

1. 分析业务边界，确定服务通信方式。这个步骤与全新设计系统时相同。
2. 冻结遗留服务。除了 bugfix，禁止在遗留系统中添加新功能。
3. 拆分新服务。剥离出遗留系统中较独立的业务，对于这块业务中的新功能统一在新服务中实现，旧功能采用代理或者委托的设计模式依然复用遗留系统。
4. 订正数据。等新服务基本承担了业务访问的时候，为新服务独立数据库，同时从旧数据库同步数据，并进行适当的数据订正。
5. 循环迭代。选择另一个业务，重复 2~4 步骤操作。

5.2 统一配置中心

一般来说，每个服务都会有很多依赖和配置信息，比如数据库、缓存、队列这样的依赖服务地址信息，以及各种运行时系统阈值和灰度发布策略参数。在一个企业级应用里，此类服务可能还会同时存在多个环境，比如开发环境、联调环境、测试环境、预发布环境、生产环境等，每个环境的配置信息又不一样。一般的做法是，对配置项类型进行划分，比如把 Spring 和数据库的配置信息独立，在每个工程目录下放一些 properties 或者 yaml 文件，用来存放配置信息，代码中涉及的依赖从这些配置文件中统一读取。对于不同的环境不同配置的需求，我们要为每个环境创建一个配置目录，该环境下的所有配置项和配置文件都存放在这个目录里，各个环境的应用在启动时使用环境变量从这些配置目录中选择相应的配置读取。

当业务扩张到一定程度，技术架构上开始服务化设计之后，服务数量的裂变会进一步加剧配置项管理的复杂性。同时还有一个问题不得不面对，即当我们需要对线上运行的应用配置项进行修改时，我们要重新走一个版本库提交、构建部署应用、重启应用的繁重流程。这个时候我们需要一个统一配置中心来帮助业务实现统一管理配置和动态更新配置的能力。

1. 实现方法

统一配置中心有多种实现思路，一种比较常见的架构如图 5-2 所示。

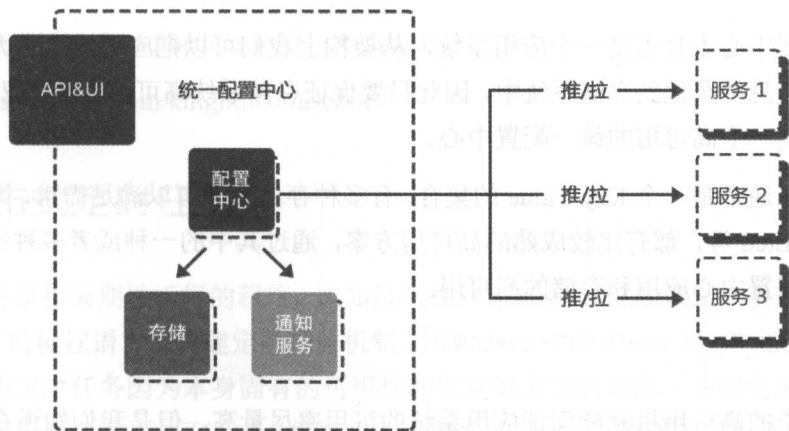


图 5-2 统一配置中心常见实现思路

配置信息通过 API 或者 UI 组件提交到配置中心，持久化到存储系统里。服务实例通过统一配置中心获取配置信息，如果配置项发生变更，配置中心通过通知服务告诉各服务实例关于配置的变更。各服务收到配置变更的通知之后，使新的配置项生效，这样就实现了应用的运行时动态更新。通知机制一般有两种实现方式。

- 服务器推。由服务器主动通知客户端配置变更。这种通知方式实时性非常好，一旦有配置更新，客户端能立刻收到新的配置信息，缺点是实现比较复杂，需要客户端与服务器端维护一个长连接，同时还要考虑网络故障可能引起的通知丢失问题。
- 客户端拉。由客户端定期拉取服务器端配置信息。它的优点是实现简单，并且因为有定期同步机制，所以相对来说可靠性较高，能够很好地容忍网络故障。但是这种通知方式实时性差些，它的延时时间由定时任务的最小间隔决定，并且因为大部分时候配置信息都是不变的，所以客户端拉模式对网络流量的消耗也是一个考虑的因素。

当所有业务的配置项都整合到统一配置中心以后，统一配置中心的可靠性会直接影响到整个系统的可用率和 SLA。为了避免统一配置中心成为系统单点故障，我们要对统一配置中心的设计添加高可用和容灾能力，这也是实现统一配置中心的难点。

2. 非功能性需求

高可用

统一配置中心本身也是一个应用系统，从架构上我们可以把应用设计为无状态，把有状态的配置信息持久化到存储系统中，因此只要保证存储系统高可用，以及应用多点部署，我们就能得到一个高可用的统一配置中心。

配置信息通常是一个 Key-Value 的集合，有多种存储系统可以满足需求，比如 MySQL、ZooKeeper、Etcd 等，都有比较成熟的高可用方案，通过其中的一种或者多种组合，我们可以实现统一配置中心应用和存储的高可用。

容灾

前面讨论的高可用机制能保证应用系统的可用率尽量高，但是我们知道在一个分布式系统里，100%的高可用是不可能实现的，特别是一些网络故障。这时我们只能通过一些容

灾策略，使得统一配置中心的失效对生产系统影响最低。以网络故障为例，系统中可能存在网络故障引起的系统失效主要有两个地方。

- 统一配置中心的应用和存储系统之间。
- 客户端和统一配置中心之间。

容灾的策略主要是缓存配置信息，比如当客户端无法连接统一配置中心时，客户端需要使用上次成功获取的配置信息，使得系统仍旧可用。换言之，此处我们认为即使是过期的配置，也比获取不到配置好。容灾能力是衡量统一配置中心是否适合生产环境使用的一个主要因素。

3. 开源系统

业界关于统一配置中心也有多个版本的开源实现。如果不是有特殊需求，建议使用这些比较成熟的实现方案。

- Spring Cloud Config: Spring 出品的配置中心方案，同 Spring 套件无缝结合。
- Disconf: 百度开源的分布式配置管理平台，提供各种配置管理的通用组件和通用平台。

有关 Spring Cloud Config 和 Disconf 的特性和具体用法可以参考以下资料。

- (1) <https://cloud.spring.io/spring-cloud-config/>。
- (2) <https://github.com/spring-cloud/spring-cloud-config>。
- (3) http://disconf.readthedocs.io/zh_CN/latest/。
- (4) <https://github.com/knightliao/disconf>。

5.3 分布式定时任务

定时任务是指周期性运行的程序，比如最常见的 Linux Crontab 就是一种单机版定时任务。很多流行的编程语言都内建定时任务机制，比如 Java 中的 Timer 类，Python 中的 sched 模块等。这类定时任务因为本身固有的可用性和性能等方面的限制，使得它们在分布式服务应用中使用场景有限。为了适应大规模分布式架构的需求，就需要引入分布式定时任务，

它要满足以下特征。

- 分布式：任务可以同时运行在多个节点中，能够调整运行的先后顺序，同时有控制并发的能力。
- 高可用：支持故障节点转移，我们需要把定时任务运行在两个节点上，这样当其中一个节点出现问题时，另外一个节点可以自动接管任务执行。
- 弹性扩缩容：每个节点分别执行分配到的任务，当节点负载达到阈值时，可以动态调整集群规模，新增节点或者下线节点对定时任务无感知。比如某一时刻任务太多或者节点故障引起集群承载能力下降，我们就需要新加节点，弹性扩容集群。
- 统一管理：定时任务的运行是分布式的，但是定时任务的管理要采用集中式的统一管理。采用统一的任务注册中心（比如数据库或者 ZooKeeper）可以方便定时任务的分配和监控。

作为一个服务化架构下的通用功能性需求，分布式定时任务大量地应用于各种场景，比如对账系统、爬虫系统定期爬取数据、定期的数据合并等。

本节主要探讨如何设计一个分布式定时任务，并介绍几款业界流行的开源框架，选用这些框架可以让业务快速获得分布式定时任务的能力。

5.3.1 分布式定时任务设计

要实现同时满足分布式、高可用、弹性扩缩容、统一管理特性的分布式定时任务，可以从抢占式或分配式两种实现方式入手。下面分别介绍它们的核心架构。

抢占式

如图 5-3 所示，每一个 Worker 都是定时任务的最终执行者，它们通过抢占的方式从调度器节点上获取任务，并定时执行。Worker 可以连上任意一个调度器节点，申请分配任务。定时任务的调度逻辑放在调度器节点，并最终持久化到任务中心。任务中心一般基于数据库或者类似 ZooKeeper 这样的协调服务实现。

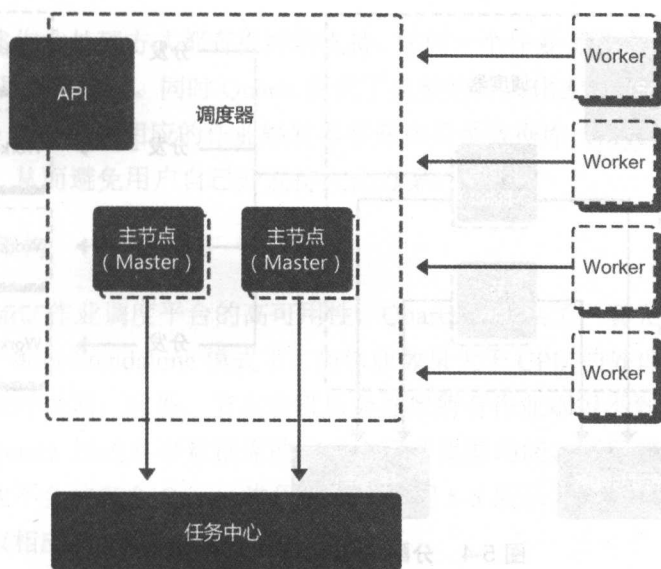


图 5-3 抢占式分布式定时任务架构

分配式

如图 5-4 所示，调度器只有一个 Master 节点，Master 节点可以通过分布式锁服务（基于 ZooKeeper 或者 Redis 等实现）选举或者竞争产生。Master 节点从任务中心获取定时任务，分配给 Worker 定时执行。当 Master 失效时，Slave 节点会重新选举出 Master，被选中的 Slave 节点成为 Master 节点，接管 Master 的工作。

在这两种方式中，用户都是通过 API 提交定时任务到系统里，具体的调度和执行由定时任务框架实现。对于抢占式调度，优点是实现简单，调度器 Master 性能可以水平扩展，缺点是它无法控制 Worker 节点的大量并行轮询，特别是在系统冷重启的时候，可能会造成任务中心的性能瓶颈。对于抢占式调度的这个缺点，我们可以通过 Worker 退避轮询或者任务中心缓存热点数据等通用方法规避或者部分解决。对于分配式调度，它的优点是可以控制任务分配的节奏，保证系统的整体负载是正常的，缺点是调度器属于一主多从模式，需要引入额外的分布式锁服务解决 Master 节点选举问题，另外，当 Master 节点性能成为瓶颈时，如果不做垂直扩容，容易造成定时任务延时，这对一些定时任务强依赖的业务系统而言不一定能接受。对于分配式调度的这个缺点，我们可以采用对定时任务分组的方式进行规避，一组调度器只负责固定数量或类型的定时任务分配。

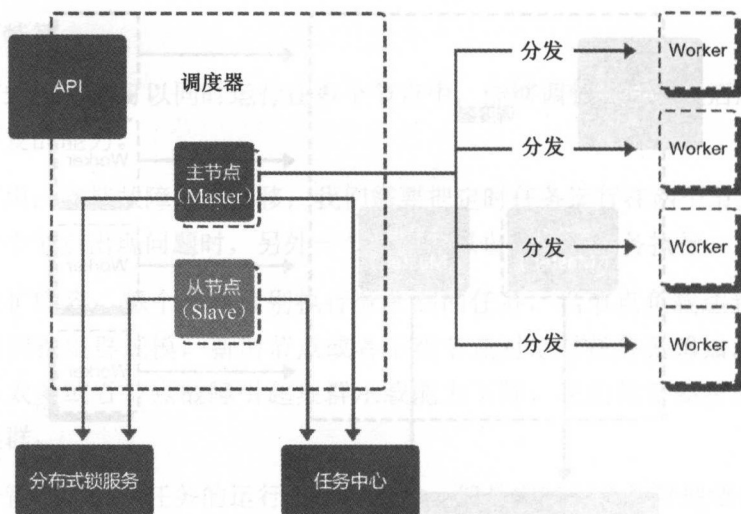


图 5-4 分布式分布式定时任务架构

5.3.2 业界流行的开源框架

1. Quartz

简介

Quartz 是 OpenSymphony 开源组织在 Job Scheduling 领域开源的项目，它既可以与 J2EE、J2SE 等应用程序相结合，也可以单独使用。Quartz 功能强大，配置灵活，可以用来创建或运行数量巨大、规模复杂的作业程序，在 Java 领域有着广泛应用，在企业应用中占重要地位。很多企业级系统选择 Quartz 也是看中它支持在集群环境中使用的能力。

Quartz 触发器主要提供以下几种作业处理方式。

1. 在一天中的某个时间点执行。
2. 在一个星期内的某一天执行。
3. 在一个月内的某一天执行。
4. 重复执行指定次数。
5. 无限重复。
6. 按一定的时间间隔重复执行。

Quartz 对上述作业处理方式都有很好的支持,任何一个作业都能实现简单的作业接口,即拥有被 Quartz 调度的能力。同时 Quartz 提供了完整的持久化方案,可以对作业持久化,通过 JdbcJobStore 把作业和相应的作业触发器存储在关系数据库(MySQL)中,以防止在重启后作业丢失,从而避免用户自己开发持久化方案。

Quartz 集群架构

为了保证 Quartz 作业调度平台的高可用性,Quartz 也提供了一套完整的故障切换、负载均衡集群方案。单点 Standalone 模式下,当作业数量大于 CPU 的处理能力时,将导致部分作业不能得到按时处理,此外,节点宕机后会导致所有作业都得不到执行,从而使相关服务受到影响。Quartz 通过共享数据库的方式实现了集群功能,采用悲观锁和标志字段等方式保证一个作业不会被多个 Quartz 进程调度。如图 5-5 所示,3 个 Quartz 进程通过与数据库直接交互获取相应的作业。

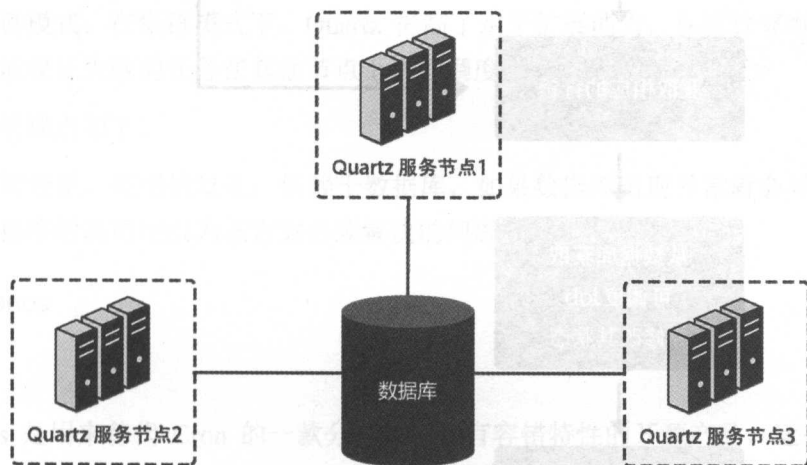


图 5-5 Quartz 集群架构

Quartz 采用“select ... for update”的方式对记录加行锁,实现对作业的调度控制,当一个 Quartz 进程获取到调度权限后,其他 Quartz 进程就获取不到该作业的调度权限。当获取到该作业的调度权限后,需要检查该作业的执行状态,如果作业在执行中或者处理成功,该作业就不需要处理,其他状态下该作业又可以继续进行调度。其具体流程如图 5-6 所示。

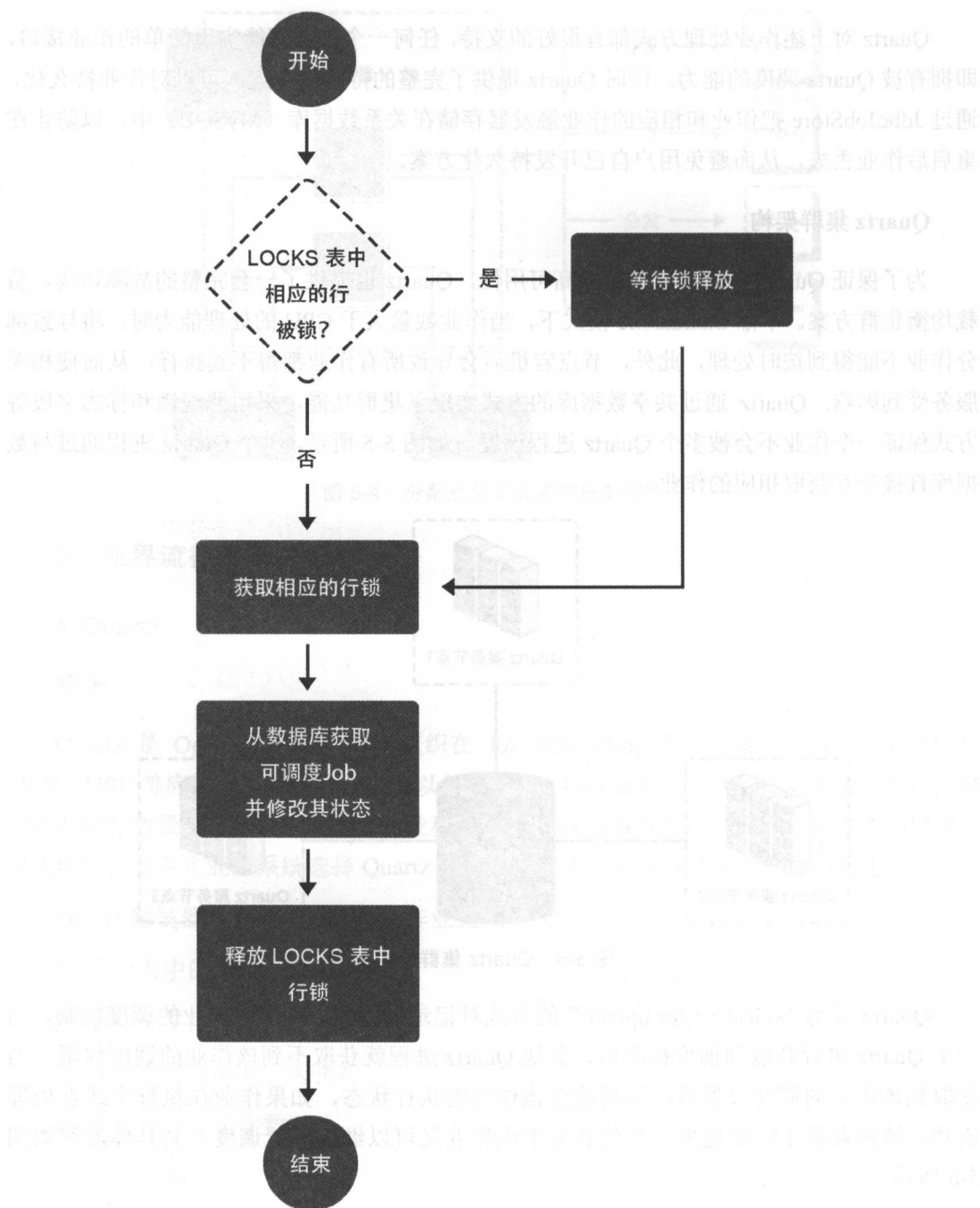


图 5-6 Quartz 集群调度流程

应用场景

Quartz 的应用场景如下。

- 定时更新数据，如定时把数据库的数据刷新到 Redis 缓存，保持缓存和数据库一致。
- 超时处理，如一个订单在指定的时间内没有结账，或者用户在指定时间还没有确认订单收货，都可以通过定时任务来完成最后业务逻辑处理。
- 一些核算业务、批处理任务、自动化任务等都可以通过 Quartz 来完成。

Quartz 的优点如下。

- 持久化作业：Quartz 通过关系型数据库进行持久化，同时通过关系型数据库能更好地对作业进行修改与配置。
- 集群模式：在集群模式下，Quartz 提高了水平扩展能力，保证计算顺畅，同时，还能保证失败的任务在其他节点上进行调度，使得作业按时执行。

Quartz 的缺点如下。

整体框架较重，使用较复杂；依赖于数据库，如果数据库出现异常就会导致整体不可用，从而数据库的高可用成为该方案必须解决的问题。

2. Chronos

简介

Chronos 是用来替代 Cron 的一款分布式、具有容错特性的开源产品，运行在 Apache Mesos 之上，可以用于作业编排。Chronos 支持自定义的 Mesos 执行器，也支持默认的命令执行器，因此，在默认情况下，Chronos 执行 sh 脚本。

Chronos 支持和系统交互，如 Hadoop，即使执行时 Mesos 的 slaves 并未安装 Hadoop。被引用的包装脚本可以在一台远程机器的后台传输并运行，并且通过异步回调来通知 Chronos 作业是否完成。Chronos 原生支持在 Docker 容器中运行作业调度。

相对于定期的计划任务，Chronos 有很多优点。Chronos 支持基于 ISO 8601 的重复时间表示法，从而能更灵活地调度作业。Chronos 也支持由其他作业的完成来触发定义。它支持任意长的依赖链。

类型

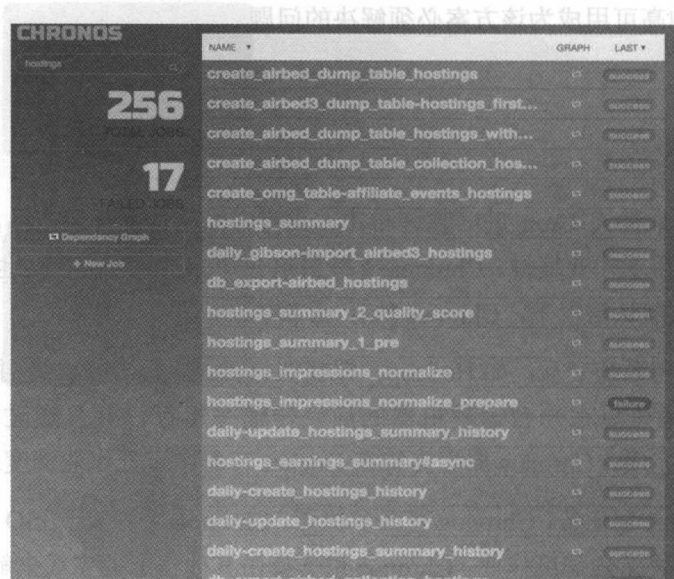
Chronos 是分配式的调度器，当有多个 Chronos 节点在运行时，只有一个会被选举为 leader。只有 leader 会处理 API 请求，如果请求发到了一个非 leader 节点，该请求会自动被转发给 leader。

核心竞争力

Chronos 支持以下特性。

- Web UI。
- 支持 ISO 8601 重复时间表示法。
- 能够处理依赖关系。
- 作业统计（如 50%、75%、95%、99%成功/失败的百分位数时间）。
- 作业的历史（如作业持续时间、开始时间、结束时间、成功/失败）。
- 默认的容错性（主从方式）。

Chronos 配有 UI 来增删改查并运行作业，也能通过图表的方式展示作业依赖，如图 5-7、图 5-8 所示。



| NAME | GRAPH | LAST |
|---|-------|---------|
| create_airbed_dump_table_hostings | CS | Success |
| create_airbed3_dump_table-hostings_first... | CS | Success |
| create_airbed_dump_table_hostings_with... | CS | Success |
| create_airbed_dump_table_collection_hos... | CS | Success |
| create_omg_table-affiliate_events_hostings | CS | Success |
| hostings_summary | CS | Success |
| daily_gibson-import_airbed3_hostings | CS | Success |
| db_export-airbed_hostings | CS | Success |
| hostings_summary_2_quality_score | CS | Success |
| hostings_summary_1_pre | CS | Success |
| hostings_impressions_normalize | CS | Success |
| hostings_impressions_normalize_prepare | CS | Failure |
| daily-update_hostings_summary_history | CS | Success |
| hostings_earnings_summary#async | CS | Success |
| daily-create_hostings_history | CS | Success |
| daily-update_hostings_history | CS | Success |
| daily-create_hostings_summary_history | CS | Success |
| db_export-airbed_collection_hostings | CS | Success |

图 5-7 Chronos 作业列表

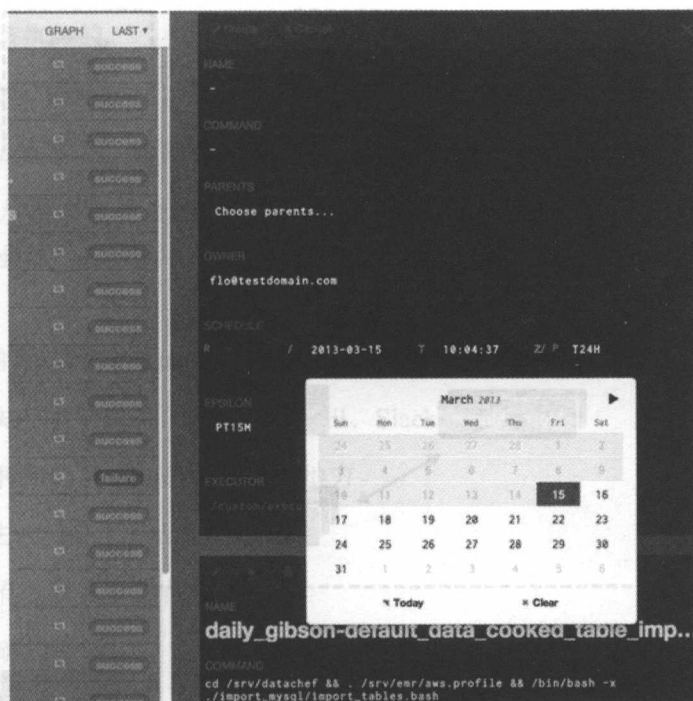


图 5-8 Chronos 作业详情

此外，Chronos 可以展示过去作业执行的统计数据，包括聚合统计，如执行成功或失败的次数。如果使用 Cassandra 集群，还可以获得每个作业执行的统计数据（也就是执行时间和状态）。

架构示例

采用 Chronos 实现的分布式定时任务设计架构如图 5-9 所示。

应用场景

Chronos 的应用场景比较多，目前应用比较广泛的是大数据处理分析。有了元数据后，我们可能需要定时处理离线数据，来完成 ETL 相关的工作。这时候可以使用传统的定时工具如 Cron 等，但是 Chronos 是一个更完善的系统，它允许重试，是轻量级的，并且提供了易用的界面展示哪些作业成功了，哪些失败了。

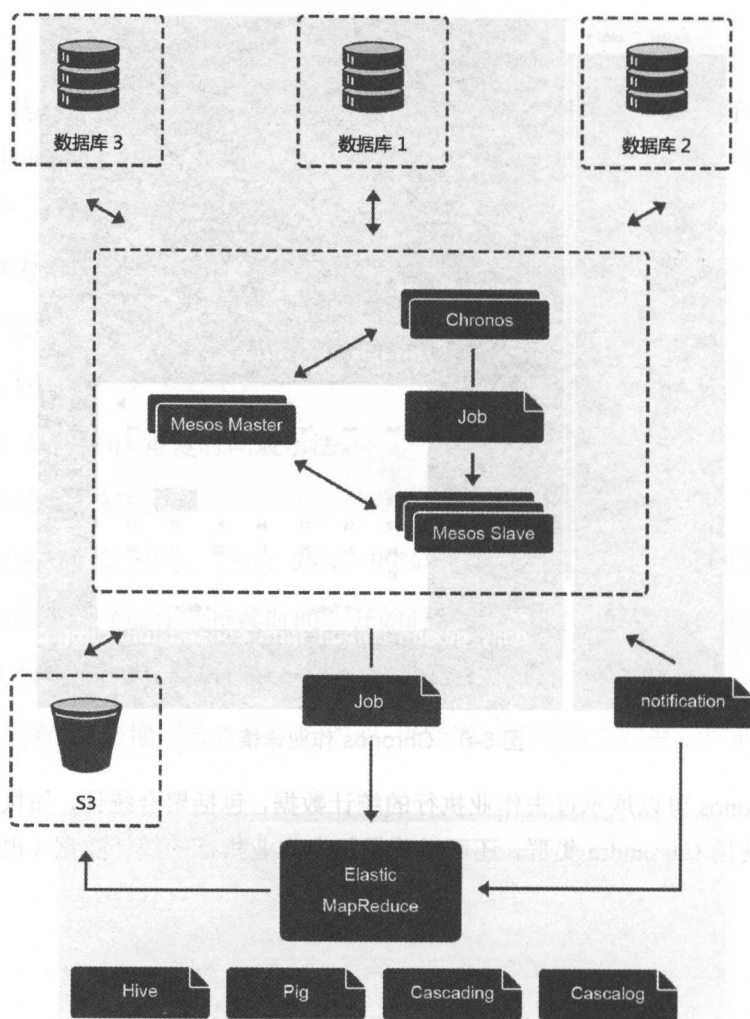


图 5-9 基于 Chronos 的分布式定时任务架构示例

如何工作

Chronos 调度器主流程比较简单，模式如下。

- 从状态存储（ZooKeeper）读取所有作业状态。
- 作业被注册到调度器中，并且加载到作业图中，用以追踪依赖。
- 作业被分为两部分，一部分为当前应该执行的（基于主机的时钟），另一部分为当

前不应该执行的。

- 要执行的作业会排队，并且在条件满足时会尽快被执行。
- Chronos 会等待直到下一个作业被调度运行，然后再从第一步开始。

此外，一个从属的作业会排队等待执行，直到它所有的父作业都成功执行完成至少一次。从属的作业执行完成后，这个执行过程会被重置。除此之外，Chronos 有一些高级特性。

- 写作业的指标到 Cassandra 中，用于后续的分析、校验及其他支持。
- 发送通知到各种各样的终端如 Email、Slack 等。
- 导出各种指标到 Graphite 及其他地方。

Chronos 无法做到以下几点。

- 解决所有分布式计算问题。
- 保证精确的调度。
- 保证时钟同步。
- 保证作业实际已运行。

3. Scheduled Job

简介

Scheduled Job 是 Kubernetes 集群用于管理基于时间的作业，即在指定的时间点执行一次或者在指定的时间点执行多次。

一个 Scheduled Job 对象就像 crontab 文件中的一行，按照 Cron 格式的计划表定期运行任务。

典型的用法是在一个指定的时间点执行一个任务或者创建一个周期性的任务，例如，数据库备份、发送邮件。

如果要使用 Scheduled Job，Kubernetes Cluster 的版本需要在 1.4 以上，并且需要在启动 API server 的时候使用 `--runtime-config=batch/v2alpha1` 开启 batch/v2alpha1 支持。

创建 Scheduled Job

一个简单的示例。每分钟运行一个 Job，打印当前时间然后“say hello”。

```
apiVersion: batch/v2alpha1
kind: ScheduledJob
metadata:
  name: hello
spec:
  schedule: 0/1 * * * ?
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

执行以下命令来运行这个示例。

```
$ kubectl create -f ./sj.yaml
scheduledjob "hello" created
```

或者，使用 `kubectl run` 创建一个不需要写完整配置的 Scheduled Job。

```
$ kubectl run hello --schedule="0/1 * * * ?" --restart=OnFailure
--image=busybox -- /bin/sh -c "date; echo Hello from the Kubernetes cluster"
scheduledjob "hello" created
```

之后，使用以下命令获取状态。

```
$ kubectl get scheduledjob hello
```

| NAME | SCHEDULE | SUSPEND | ACTIVE | LAST-SCHEDULE |
|-------|-------------|---------|--------|---------------|
| hello | 0/1 * * * ? | False | 0 | <none> |

可以看到，既没有活动的 Job 也没有被调度的 Job。等待时间大约 1 min，Job 被创建出来。

```
$ kubectl get jobs --watch
NAME          DESIRED  SUCCESSFUL  AGE
hello-4111706356  1        1          2s
```

现在能看到有一个运行中的作业，该作业由“hello”调度。我们可以停止它然后重新获取 Scheduled Job。

```
$ kubectl get scheduledjob hello
NAME          SCHEDULE          SUSPEND  ACTIVE  LAST-SCHEDULE
hello         0/1 * * * ?      False   0       Mon, 29 Aug 2016 14:34:00 -0700
```

在 LAST-SCHEDULE 这个时间点“hello”成功调度了一个作业。当前有 0 个活动的 Job，表示 Job 的调度已经完成或失败了。找出由最近调度任务创建的 Pod 并且观察其中一个 Pod 的标准输出。请注意 Job 名和 Pod 名是不同的。

```
# Replace "hello-4111706356" with the job name in your system
$ pods=$(kubectl get pods --selector=job-name=hello-4111706356
--output=jsonpath={.items..metadata.name})
$ echo $pods
hello-4111706356-o9qcm
$ kubectl logs $pods
Mon Aug 29 21:34:09 UTC 2016
Hello from the Kubernetes cluster
```

删除 Scheduled Job

一旦不再需要某个 Scheduled Job，可以通过 kubectl 命令删除它。

```
$ kubectl delete scheduledjob
hello scheduledjob "hello" deleted
```

该命令会停止已经创建出来的作业，但是正在运行中的作业不会被停止，并且 Job 和 Pod 不会被删除。如果要清理这些 Job 和 Pod，需要列出来由 Scheduled Job 创建出来的所有 Job，然后全部删除。

```
$ kubectl get jobs
```

```
NAME                DESIRED  SUCCESSFUL  AGE
hello-1201907962    1        1           11m
hello-1202039034    1        1           8m ...
$ kubectl delete jobs hello-1201907962 hello-1202039034 ...
job "hello-1201907962" deleted job "hello-1202039034" deleted ...
```

一旦 Job 被删除，由它创建出来的 Pod 也会被删除。请注意所有由“hello”创建出来的 Job 都使用前缀“hello-”。如果想删除当前 namespace 下的所有 Job（不只由“hello”创建），你可以通过 `kubectl delete jobs --all` 命令全部删除它们。

Scheduled Job 的局限性

一个 Scheduled Job 一次执行期间大约创建一个 Job，这里说“大约”，是因为在特定情况下可能会创建两个 Job，或没有 Job 被创建。Kubernetes 官方正在试图使这些情况极少发生，但是不能保证完全杜绝。因此，Job 应该被设计为幂等的。

5.4 分布式锁系统

产品中经常会出现多个进程竞争同一个资源的情况。比如有一个秒杀的场景，为了提升网站活跃度，现在推出 1 元抢拍宠物的活动，这就涉及多个客户端同时查询库存的操作，为了避免出现多个用户同时抢到宠物的情况，我们需要一个锁服务对客户端和资源进行协调。

对于一个单机应用，我们可以利用编程语言内建的锁机制进行同步，比如 Java 中的 `Synchronized` 和 `java.util.concurrent.locks` 包。但是在分布式环境下，这种单机模式的锁机制就行不通了，因为编程语言内置支持的锁机制无法作用于多个进程。这时候就出现了分布式锁的需求。跟单机环境的锁机制一样，分布式锁也需要满足以下几点特性。

- 互斥。这是锁服务的基本能力，多个节点下必须确保同一时刻只有一个节点能获得锁。
- 避免死锁。死锁在分布式环境下很容易出现，比如某个节点得到锁之后，因为节点故障宕机，或者节点和锁服务之间网络故障，都会造成锁没有机会释放的情况。因为锁未释放，所以其他节点也就无法再获得锁提供正常服务，从而使得整个系

统处于死锁状态。分布式锁服务要做的是从锁本身出发，规避死锁。一般可以通过对锁加自动过期时间解决。

- 高性能。因为锁服务是用来协调分布式系统中各节点的，所以很容易形成系统性能瓶颈和单点故障。性能也是分布式锁服务是否适合生产环境的重要考虑因素。
- 可重入性。如果同一个节点可以重复获取一个已经获取到的锁，这种锁一般称为可重入锁。根据业务场景，我们可以按需求设计锁服务是否支持重入性。

分布式锁实现

根据上述的分布式锁特性，我们可以抽象出分布式锁实现模型，如图 5-10 所示。

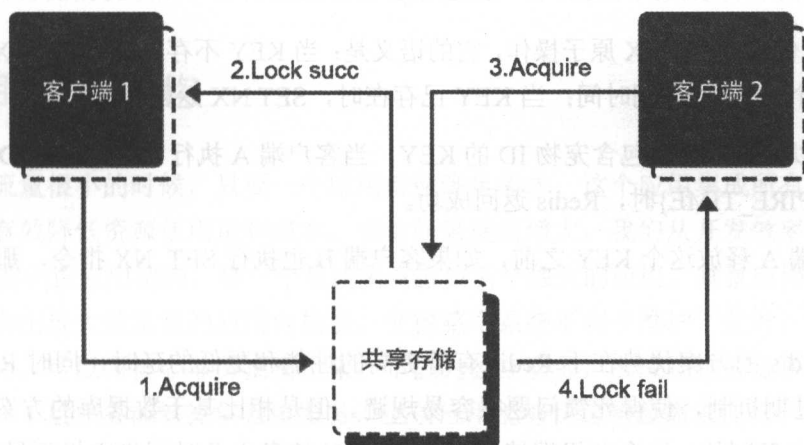


图 5-10 分布式锁实现模型

共享存储负责锁和对应客户端信息的存储。客户端 1 发起 `acquire()` 调用后，共享存储返回获取锁成功，当客户端 2 再发起对同一个锁的 `acquire()` 调用时，共享存储会返回获取锁失败。下面我们以秒杀场景为例，列举几种常见的分布式锁服务的实现。

1. 基于数据库

关系数据库（比如 MySQL）自身支持数据的 ACID 强一致性和基于复制的高可用性，我们可以利用这两个特性，为系统中的竞争资源设计一张 `lock` 表。这张表用一列表示资源 ID，表示正在尝试分配的资源，这一列用 `UNIQUE KEY` 约束，通过数据库保证这个资源 ID 不会被尝试分配多次，以此来提供锁能力，保证客户端互斥执行。

当客户端 A 尝试拍下宠物时, 先将这只宠物 ID 加入到 lock 表, 然后再进行剩余的支付操作, 等支付成功后, 将 lock 表中记录删除。如果在记录删除之前, 有客户端 B 尝试拍下这只宠物, 因为得不到锁, 所以会返回给客户端 B 一个失败提醒。

上述方案实现的分布式锁拥有较高的可靠性和稳定性, 但是性能会差些, 并且会存在死锁问题。一旦客户端 A 在获得锁之后故障宕机, 会导致数据库中的锁记录一直无法释放, 那么其他客户端也就一直无法拍下宠物。解决这个问题需要为锁记录增加一个自动过期的机制, 比如基于数据库的方案可以采用定时任务定期删除超过一定时间的锁记录。用相同的思路基于 Redis 缓存实现是一种更加优雅的方案。

2. 基于 Redis 缓存

Redis 提供 SET NX EX 原子操作, 它的语义是: 当 KEY 不存在时, SET NX 返回成功, 并且设置这个 KEY 的过期时间; 当 KEY 已存在时, SET NX 返回失败。

我们可以构造一个以包含宠物 ID 的 KEY, 当客户端 A 执行 SET {PET_ID} {VALUE} NX EX {EXPIRE_TIME} 时, Redis 返回成功。

在客户端 A 释放这个 KEY 之前, 如果客户端 B 也执行 SET NX 指令, 那么 Redis 会返回失败。

基于 Redis 的方案优势在于 Redis 有着更高的性能和更低的延时, 同时 Redis 原生支持的 KEY 过期机制, 使得死锁问题很容易规避。但是相比基于数据库的方案, Redis 的高可用是异步复制的, 这会在极端情况下出现 KEY 信息未及时同步宕机而导致锁丢失的情况。

上述方案本质上还是基于单点的 Redis 设计分布式锁, 满足了大部分场景的需求, 而 Redis 作者也曾提供一种基于 N 个独立 Redis 节点的分布式锁 Redlock 的实现。

3. 基于 ZooKeeper

ZooKeeper 天生就是提供分布式协调服务的, 非常适合用来实现分布式锁服务。有以下几个原因。

- ZooKeeper 实现了 Paxos 一致性协议, 从协议层面支持多节点的数据一致性问题, 能容忍网络分区。

- ZooKeeper 的临时节点支持客户端断连后自动删除。这个特性可以用来保证客户端异常后锁自动释放，规避死锁问题。
- ZooKeeper 支持 Watcher 机制。当锁被删除或者释放之后，其他客户端可以实时收到通知。

我们用一个包含了宠物 ID 信息的 `znode/path-to-lock/{pet-id}` 表示锁。多个客户端节点同时请求这个节点，只有一个节点能返回成功，其余返回失败。如果未抢到锁的客户端关心锁释放问题，可以同时注册一个 Watcher 事件，等待 ZooKeeper 的通知。

本节主要讨论了分布式锁的实现原理及几种实现方案。每种方案都有在可靠性、可用性、易用性等方面的优缺点。如何根据业务场景选型，需要技术人员分析并验证。

5.5 微服务化架构

在网站流量很小的时候，只要一个应用就能满足需求，这个应用集成所有功能，统一部署，能够有效降低资源使用量和成本。当访问量逐渐增大，我们从开发效率考虑，选择基于 MVC 的垂直应用架构，将一个系统垂直拆成多个独立的应用。垂直应用架构有一个问题，就是会出现大量重复的应用内模块，使得整个系统不利于维护。此时，为了提高代码和功能复用，可以将核心业务抽离出来，下沉为独立服务，形成稳定的服务中心。经过这样的改造，下层服务能够提供稳定服务，上层应用能够快速响应需求变化。

例如我们有多个独立的 MVC 架构应用——支付应用、订单应用、评价应用等，现在要统一在各个应用中加入流控模块。流控本身业务并不复杂，只需统计用户访问量，当单位时间内访问量达到阈值时，返回 API 调用失败。在服务化之前，我们只能在每个独立应用中添加流控代码，然后每个应用独立回归，一处修改，到处回归。如果把流控独立成微服务，所有应用在需要流控的地方或者拦截器里统一调用流控服务，就能很大程度上避免全量回归的问题。这就是微服务化带来的收益。

微服务是单一应用程序构成的小服务，自己拥有自己的进程与轻量化处理，通过业务功能确定服务边界，以自动化的方式部署，与其他服务之间通过 HTTP API 通信，不同的微服务可以用不同的编程语言和技术框架实现。

微服务化是当前互联网产品的一个技术架构趋势，近两年伴随着 Docker 容器技术的发

展和普及，微服务架构在业界逐渐落地。那么驱动企业进行微服务化架构改造的根本推动力是什么？如何在企业内部实施微服务架构？实施微服务架构需要哪些技术框架和基础设施？这些都是企业技术人员需要了解并关心的问题，也是本节要具体讨论的问题。微服务作为对应单体应用出现的概念，其架构通常有哪些优势呢？

- 项目工程简洁。一个复杂的产品功能集合，代码仓库规模往往随着业务复杂度的增加而线性增加。对于单体应用来说，代码的堆积意味着工程规模的迅速膨胀。而对于微服务来说，因为每个微服务承担的职责小而而且单一，所以工程规模简洁可控。
- 升级代价小。当一个产品所有的功能都集中在同一个应用时，对程序的升级会带来两方面的影响：一是项目工程过大造成的应用启动时间过久，我们见过有些产品一个应用实例的启动时间需要半个小时以上，这对业务来说显然不可行。二是每次 hotfix 都需要对整个应用重启，引起业务的不稳定。而微服务架构恰恰相反，每个服务独立升级，对业务整体的影响极小。
- 扩展性好。对于互联网产品来说，产品迭代速度很重要。对于一个庞大的单体应用来说，牵一发而动全身，无论对产品功能的扩展性还是性能的扩展性来说，都是一个很大的挑战。通过微服务化，把业务中扩展性差的部分独立出去，不同的业务类型采用不同扩展方案，可以提升业务整体的可扩展性。
- 稳定性好。单体应用的稳定性差主要体现在功能间的隔离性，对于单体应用而言，所有的功能都在同一个进程空间里，这意味着任何一个功能的 bug 可能会造成应用整个崩溃。微服务的好处是可以实现进程级别的隔离，单个服务异常很少会造成全局故障。
- 人员变更影响小。项目中人员更迭并不少见，单体应用的交接要求被交接人员必须对整个项目非常熟悉，才有可能消化变更人员带来的负面影响，否则人员离职或转岗会影响项目的正常进度。实施微服务架构的团队往往同时也遵循“2 pizza”原则的组织架构，团队小而精，人员变更影响小。
- 技术栈丰富。单体应用因为都跑在同一个进程里，所以项目的整体技术栈就被这个进程锁死了。比如说一个 Java 单体应用，无论业界的其他编程语言和开发框架如何发展，我们也无法利用起来，无法实现技术反哺业务。而微服务可以采用一些通用的服务间通信方式（HTTP 等）去集成，使得服务的实现方式不局限于某

个技术栈里，适合用最合适的技术实现功能。

- 开发效率高。主要体现在微服务架构下项目的学习曲线平滑，因为工程规模小，所以开发人员能很快做到对项目有一个大而全的认识。

选择如图 5-11 所示的微服务化作为架构演进方向之后，接下来要考虑的就是如何实施微服务架构，如何将微服务化落地，真正为产品交付带来收益。

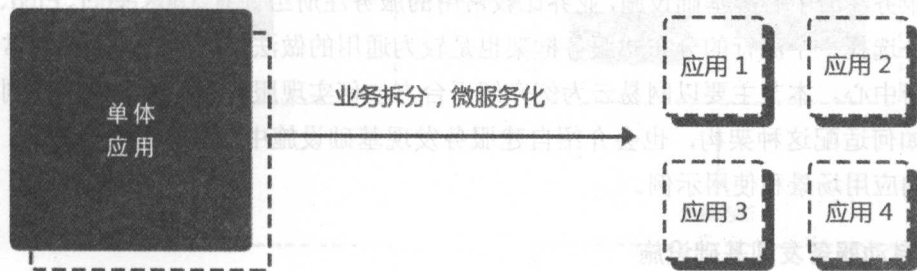


图 5-11 单体应用微服务化

本节接下来的部分将重点讨论微服务化的实施问题；单体应用在完成业务拆分之后如何进行服务的发现和治理；目前业界几款流行的微服务化实施框架如何选型；服务编排这一全新的微服务交付方式；在微服务架构下如何进行测试等。

5.5.1 服务发现

设想一下，当我们实施了微服务架构以后，原本的会员、购物车、订单、评论、支付模块等都是独立服务的形式部署，对外提供统一的 HTTP API 或者二进制 API。假如购物车服务需要订单服务的 HTTP API 时，购物车服务首先需要知道订单服务在网络中的位置（一般用 IP+端口号标识），如果这个网络位置是固定的，我们可以在购物车服务的配置文件中直接指定地址。但是对基于云平台或者采用云原生架构的应用来说，情况会复杂很多，因为频繁的扩缩容和更新升级等操作，服务的网络位置可能动态变化。这个时候就需要有一个服务发现机制，能够动态感知服务访问地址的变化。

服务发现机制有服务端发现和客户端发现两种实现方式，两者的主要区别在于谁负责维护服务访问地址的变化。服务端发现模式可以通过 DNS 或者带 VIP 的负载均衡实现，它的优点是对客户端无侵入性，客户端只需简单地向负载均衡或者服务域名发起请求，无需关心服务发现的细节，也不必引入编程语言和框架强相关的服务发现逻辑。而对于客户端

发现模式来说，客户端需要从一个服务注册表中查询服务地址列表，再决定通过哪个地址请求服务。客户端发现模式虽然引入了服务注册表的依赖，但是得到了更多灵活性，同时，客户端可以通过汇聚业务信息来进行更有效的负载均衡。

一般来说，云原生应用架构的平台方会默认提供服务端的服务发现机制，对客户端透明，比如 AWS 和网易云都提供了相应的解决方案。如果你选择了客户端发现模式，就要自己搭建服务注册中心等基础设施，业界比较常用的服务注册组件有 ZooKeeper、Etcd、Consul 等。另外选择一个流行的分布式服务框架也是较为通用的做法，这类服务框架通常会内置服务注册中心。本节主要以网易云为例介绍平台方如何实现服务端的服务发现机制，以及客户端如何适配这种架构，也会介绍自建服务发现基础设施中 ZooKeeper、Etcd、Consul 等组件的应用场景和使用示例。

1. 自动服务发现基础设施

目前越来越多的微服务技术被使用在云平台部署应用和服务中，从而催生了更多细小服务单元的出现。为了满足各个服务之间或者服务内部的资源访问，服务端服务发现被集成于大多数云平台中。下面以网易云平台为例介绍如何实现一套服务端服务发现机制。

网易云平台的服务发现

网易云平台提供两种服务发现机制：环境变量和 DNS 服务。

- 环境变量：创建微服务时为服务容器实例设置被访问服务的 HOST IP 地址和 PORT 端口等环境变量。容器通过环境变量解析被访服务地址，从而访问到目标服务。
- DNS：通过 DNS 服务组件，为集群内部主机或容器提供解析地址服务。新建微服务时通过服务发现写入 DNS 记录，并在 DNS 服务组件中缓存数据。如果 DNS 在整个集群范围内可用，那么所有容器实例都能够自动解析服务的域名。即只需要通过域名即可访问到该服务。

环境变量在创建容器时需要配置，如果容器要访问多个服务，必须为容器创建多个环境。如果服务名发生改变，修改多个容器的多个环境变量，很容易带来问题，维护起来更复杂。大多数业务使用 DNS 的方式作为服务端服务发现。

网易云基于 Kubernetes 的服务发现架构

网易云服务发现架构如图 5-12 所示。

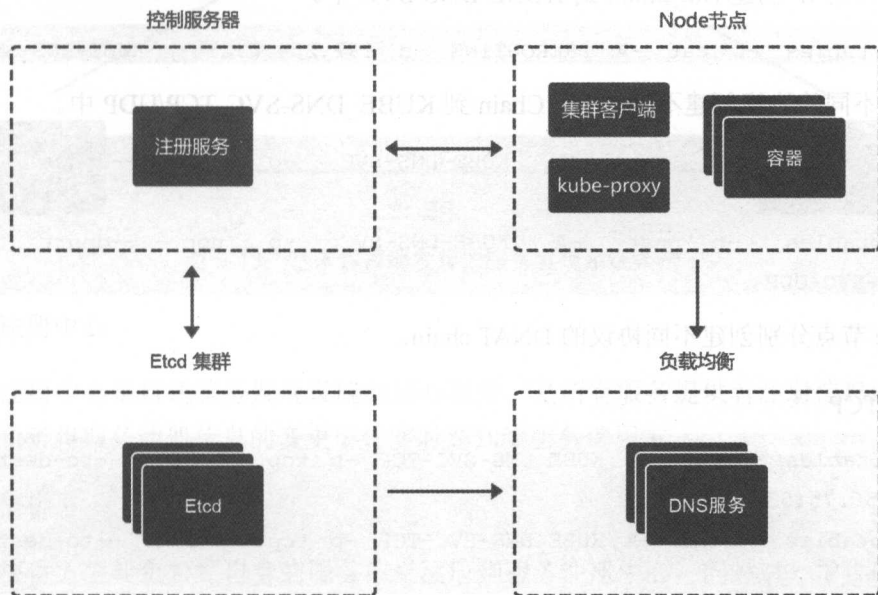


图 5-12 网易云基于 Kubernetes 的服务发现架构

- 控制服务器提供服务注册能力，服务配置数据存储到 Etcd 集群中。
- DNS 服务集群通过 Etcd 事件变化，读取服务配置缓存到内存，加速 DNS 访问。多个 DNS 服务实现负载均衡。
- Etcd 集群持久化 Kubernetes 数据。
- 节点集群客户端配置 DNS 集群地址。容器通过 DNS 地址解析到服务地址，从而访问服务。Node 节点能访问 DNS 服务目前依赖 kube-proxy 服务组件，kube-proxy 服务组件为每个容器创建访问 DNS 服务的 Iptable 规则。

我们以两个副本为例介绍 iptables 规则设置：service IP 为 192.254.0.2，副本 IP 分别为 10.180.156.75，10.180.156.78。

Node 节点分别创建 3 个 Nat Chain。

```
iptables -t nat -N KUBE-DNS-SVC
```

```
iptables -t nat -N KUBE_DNS-SVC-TCP
iptables -t nat -N KUBE_DNS-SVC-UDP
```

根据目的 IP 创建 Nat Chain 到 KUBE-DNS-SVC 中。

```
iptables -t nat -A PREROUTING -d 192.254.0.2 -j KUBE-DNS-SVC
```

根据不同的协议创建不同的 Nat Chain 到 KUBE_DNS-SVC-TCP/UDP 中。

```
iptables -t nat -A KUBE-DNS-SVC -p tcp --dport 53 -j KUBE_DNS-SVC-TCP
iptables -t nat -A KUBE-DNS-SVC -p udp --dport 53 -j KUBE_DNS-SVC-UDP
```

Node 节点分别创建不同协议的 DNAT chain。

● TCP

```
iptables -t nat -A KUBE_DNS-SVC-TCP -p tcp -j DNAT --to-destination 10.180.156.75:53
iptables -t nat -A KUBE_DNS-SVC-TCP -p tcp -j DNAT --to-destination 10.180.156.78:53
```

● UDP

```
iptables -t nat -A KUBE_DNS-SVC-UDP -p udp -j DNAT --to-destination 10.180.156.75:53
iptables -t nat -A KUBE_DNS-SVC-UDP -p udp -j DNAT --to-destination 10.180.156.78:53
```

这里需要注意的是，针对同一个协议有两个目标容器，可以根据实际的需要给任意的目标容器配置 `-m statistic --mode random --probability` 匹配率，这样可以避免同时向两个目标发送相同的请求，同时相当于一个简单的负载均衡。

2. 自建服务发现

我们先来看一下基于客户端发现的服务发现系统架构，如图 5-13 所示。

该架构中有 3 种角色：服务注册中心、服务提供方和服务调用方。

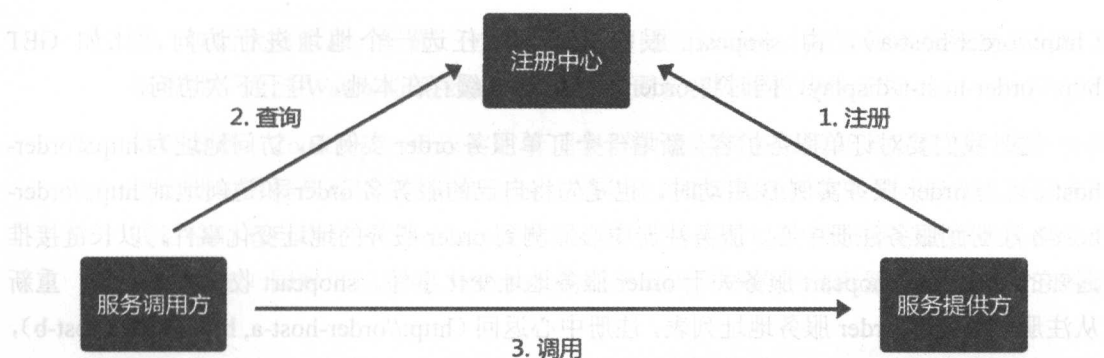


图 5-13 基于客户端发现的服务发现系统架构

服务注册中心

服务注册中心是自建服务发现设施的核心组件，是所有服务提供者注册信息的存储中心，同时负责将服务注册信息的变更事件实时通知给服务调用方。

服务提供方

服务提供方主要负责将自身的服务信息注册到服务注册中心，同时对外提供服务。服务信息中最重要的信息就是服务的访问地址，通常是 IP+端口号，也可以是服务域名。

服务调用方

服务调用方作为服务发现系统中的客户端，主要职责如下。

- 服务调用方启动时从服务注册中心获取需要调用的服务注册信息，缓存在本地。
- 根据本地缓存中的服务注册信息构造服务调用请求，并根据配置的负载均衡策略发出服务调用请求。
- 接收服务注册中心关于服务注册信息变更的通知，更新本地缓存中服务提供方的注册信息。

以购物车服务 `shopcart` 和订单服务 `order` 为例，`shopcart` 服务需要调用 `order` 服务的 HTTP API `GET/display` 来展示订单详情。`order` 服务启动时，首先将当前实例 A 的服务名 `order` 和访问地址 `http://order-host-a` 注册到服务注册中心，`shopcart` 要调用 `order` 服务时去注册中心以服务名 `order` 为参数获取 `order` 服务地址，注册中心返回 `order` 服务地址列表

(`http://order-host-a`)，由 `shopcart` 服务从列表中任选一个地址进行访问，比如 `GET http://order-host-a/display`，同时将 `order` 服务的地址缓存在本地，用于下次访问。

这时我们要对订单服务扩容，新增一个订单服务 `order` 实例 B，访问地址为 `http://order-host-b`。当 `order` 服务实例 B 启动时，也是先将自己的服务名 `order` 和访问地址 `http://order-host-b` 注册到服务注册中心。服务注册中心监测到 `order` 服务的地址变化事件，以长链接推通知的方式告诉 `shopcart` 服务关于 `order` 服务地址变化事件。`shopcart` 收到事件通知，重新从注册中心获取 `order` 服务地址列表，注册中心返回 (`http://order-host-a`, `http://order-host-b`)，`shopcart` 服务缓存 `order` 服务新的访问地址，下次访问 `order` 服务就可以从列表中任取一个地址进行访问。

这个例子演示了一个简单的服务发现 workflow，实际在应用服务发现系统时还要考虑服务提供方的健康检查、针对服务调用的流控等，不同类型的注册中心也有不同的功能特性。本节会介绍 ZooKeeper、Etcd、Consul 在服务发现中的应用，方便读者在自建服务发现系统选型时进行参考。

(1) ZooKeeper

ZooKeeper 简介

ZooKeeper 是一个高可用高性能的分布式应用协调服务，其提供了名字空间服务、分布式应用配置管理、集群管理、锁服务等功能给开发者，从而释放了开发者的压力，无需担心分布式环境下的复杂性。ZooKeeper 框架由雅虎开发并开源，后续被 Apache 录入，被 Hadoop、Hbase 等其他分布式框架使用，常见的比如 Hbase 使用 ZooKeeper 来跟踪数据的状态，同时也被 Dubbo、disconf 等国内开源软件使用。ZooKeeper 提供的功能主要包括以下几点。

- 名称空间服务：ZooKeeper 提供的名称空间与标准文件系统类似。名称是以斜杠 (“/”) 分隔的路径元素序列。ZooKeeper 的名称空间中的每个 `znode` 都有路径标识，并且每个 `znode` 都有一个父对象，它的路径是 `znode` 的前缀，少了一个元素。此规则的例外是 `root` (“/”)，它没有父级。此外，与标准文件系统完全相同，只要 `znode` 有任何子项，就无法删除。
- 配置管理服务：ZooKeeper 中的每个 `znode` 都可以有与之相关联的数据，`znode` 节点数据存储在内存中，它们存储的数据量较少。ZooKeeper 设计用于存储协调数

据,包括状态信息、配置、位置信息等。开源的 `disconf` 配置管理系统就是利用了该特性,通过 `Watch` 监视 `znode` 上的数据更新实现配置更新下发。

- **集群管理:** `ZooKeeper` 可以实时管理集群中机器的上下线及节点状态,每一台机器启动的时候都可以在 `ZooKeeper` 上进行 `EPHEMERAL` 临时节点注册,当节点下线后临时节点将会被 `ZooKeeper` 自动删除,从而可以动态地观察集群中机器的上下线情况,同时可以通过 `Watcher` 机制获取到具体的下线机器从而进行相应的运维处理。
- **Leader 选举:** 在分布式环境下,相同的业务或者服务往往会部署在多台机器上,但是部分服务只希望有一台机器能够执行,以避免多台机器或者进程执行导致的数据不一致、重复执行等问题,使用 `ZooKeeper` 提供创建 `znode` 的唯一性保证同一时间只有一个服务能够创建该 `znode`,从而保证只有一个业务或者服务能够得到执行。
- **锁服务:** 锁服务依赖于 `ZooKeeper` 提供的强一致性,锁的形式有两种,一种是独占的方式,同一时刻只有一个服务能够成功获取到这个锁,另一种是时序控制型锁,获取这个锁的客户端都能成功获取到锁,只是获取锁是存在时序的,通过 `EPHEMERAL_SEQUENTIAL` 来实现。

ZooKeeper 集群结构

`ZooKeeper` 集群由一组 `Server` 节点组成,这组 `Server` 节点存在两个角色,分别为 `Leader` 和 `Follower`,同一时刻整个集群只有一个 `Leader` 角色,其余都为 `Follower`。为保证 `Leader` 选举得到多数支持,通常 `Server` 节点的个数是 $2n+1$ 个,当可用节点大于 n 时,整个 `ZooKeeper` 集群才可用。客户端连接到 `ZooKeeper` 集群,并且执行事务请求时,为保证强一致性,这些请求会被转发送到 `Leader` 节点上, `Leader` 将客户端的事务请求转换为事务 `Proposal`,并且将 `Proposal` 事务分发给集群中其他所有的 `Follower` 并等待投票,当有超过半数的 `Follower` 投票通过后, `Leader` 将再次向集群内 `Follower` 广播 `Commit` 提交信息, `Commit` 将之前的 `Proposal` 事务提交,从而保证了事务的原子特性。如图 5-14 所示。

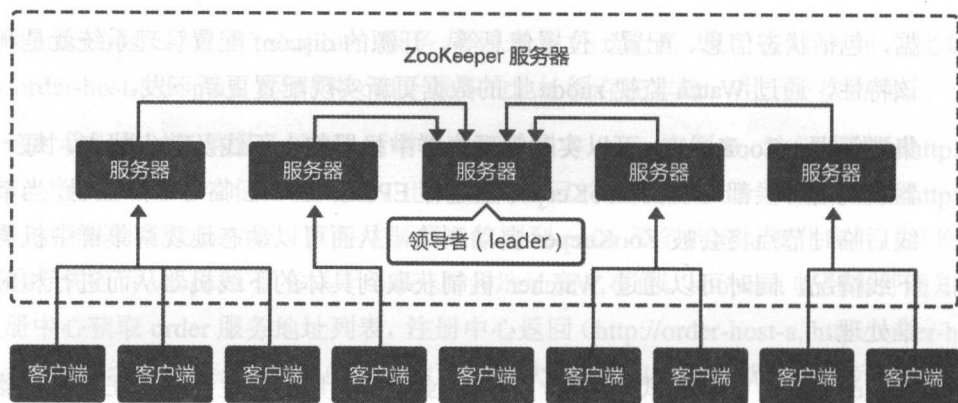


图 5-14 ZooKeeper 集群结构

ZooKeeper 使用了一种自定义的原子消息协议 ZAB, 该算法在 Paxos 算法基础上进行了扩展, 增加了支持崩溃恢复等特性。ZooKeeper 采用 ZAB 协议将服务器状态以原子消息形式广播到所有 Follower 上, 以此保证整个系统中的节点数据或状态一致, 同时 Follower 节点基于这种消息协议能够保证本地数据与 Leader 节点同步。由于在分布式环境下节点异常是很常见的, 为了保证高可用性, 当集群中 Leader 节点发生故障失效时, ZooKeeper 通过 ZAB 协议能够快速选择一个新的 Leader 提供服务, 继续处理客户端的事务请求。

基于 ZooKeeper 服务注册与发现

通过 ZooKeeper 提供的相应功能, 我们可以在其基础上实现完整的服务发现机制, 能够方便地处理服务注册、监听等事件, 下面通过一个具体实例来说明如何使用 ZooKeeper 实现多个服务之间的注册与服务发现。

在会员中心、购物车、订单、评论这样一个电商场景中, 各个服务之间是存在依赖关系的, 比如订单服务要访问购物车和用户的会员信息。在服务化部署后, 购物车和会员中心需要暴露相应的接口以便订单服务进行调用。同时在电商场景下往往存在大量的促销活动, 为应对高并发大流量, 服务水平扩缩容也是服务必须设计考虑的一部分。那么在服务扩缩容后, 其他依赖服务怎么来进行感知呢? 如图 5-15 所示的各个服务在启动后都在 ZooKeeper 上注册, 通过 ZooKeeper 来管理整个系统的状态, 各个服务只需要查询 ZooKeeper 即可获得其依赖服务的最新信息, 或者通过 Watcher 机制动态地获取依赖方的更新。具体设计如图 5-16 所示, 在 ZooKeeper 的根目录下创建一个基于服务而划分的树状目录结构,

其中/Company/Services/作为父目录，所有的服务都在这个目录下进行注册。任何一个服务会有生产者和消费者，其中生产者就是注册方，而消费者就是依赖方，生产者在注册时提供了服务部署的详情，在 **Producer** 节点下可以注册多个生产者（每个服务会部署在多个节点上，既可实现高可用又可以保证负载均衡）。如 **ServiceN** 这个节点保存了第 **N** 个生产者的具体部署信息，包括 **IP** 及端口和权重等信息。而 **Consumer** 保持了每个服务依赖方的使用信息，用于对服务的管理和维护。

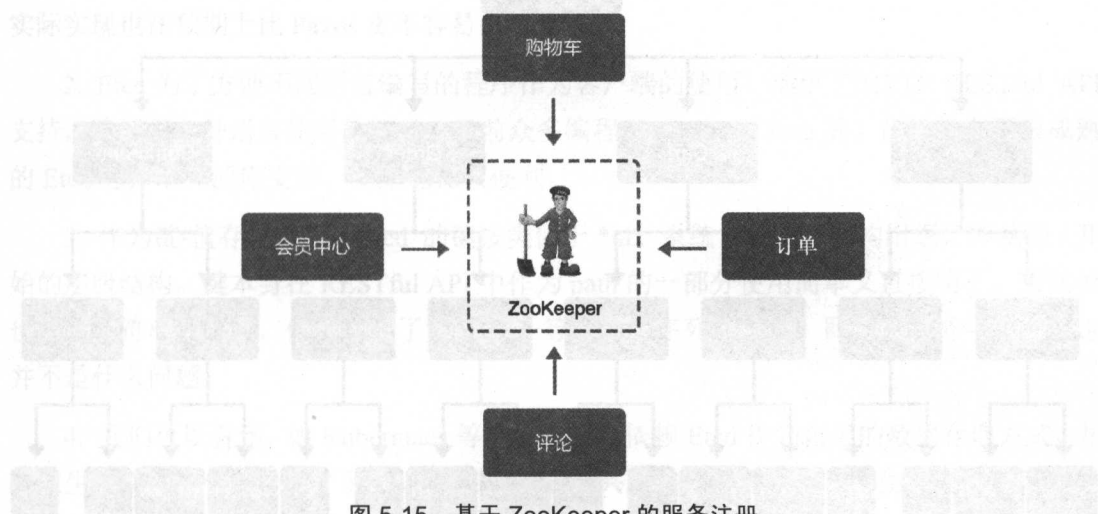


图 5-15 基于 ZooKeeper 的服务注册

如图 5-16 所示，整个服务发现的具体流程如下。

1. 注册会员服务，在/Company/Services/目录下创建/Company/Services/Vip/Producers 和 /Company/Services/Vip/Consumers 这两个永久节点。
2. 会员服务上线，在/Company/Services/Vip/Producers 下创建具体 Service 临时节点，每个 Service 临时节点保存了相应服务的具体信息。
3. 订单或者其他服务需要访问会员服务，先在/Company/Services/Vip/Consumers/下增加一个消费者用于记录调用与被调用的关系。
4. 订单或者其他服务通过 ZooKeeper 客户端获取/Company/Services/Vip/Producers 目录下的具体部署信息，从而实现服务发现。
5. 当会员 VIP 服务进行扩缩容时，订单或者其他服务通过 ZooKeeper 客户端监视 /Company/Services/Vip 下的变化更新本地路由。

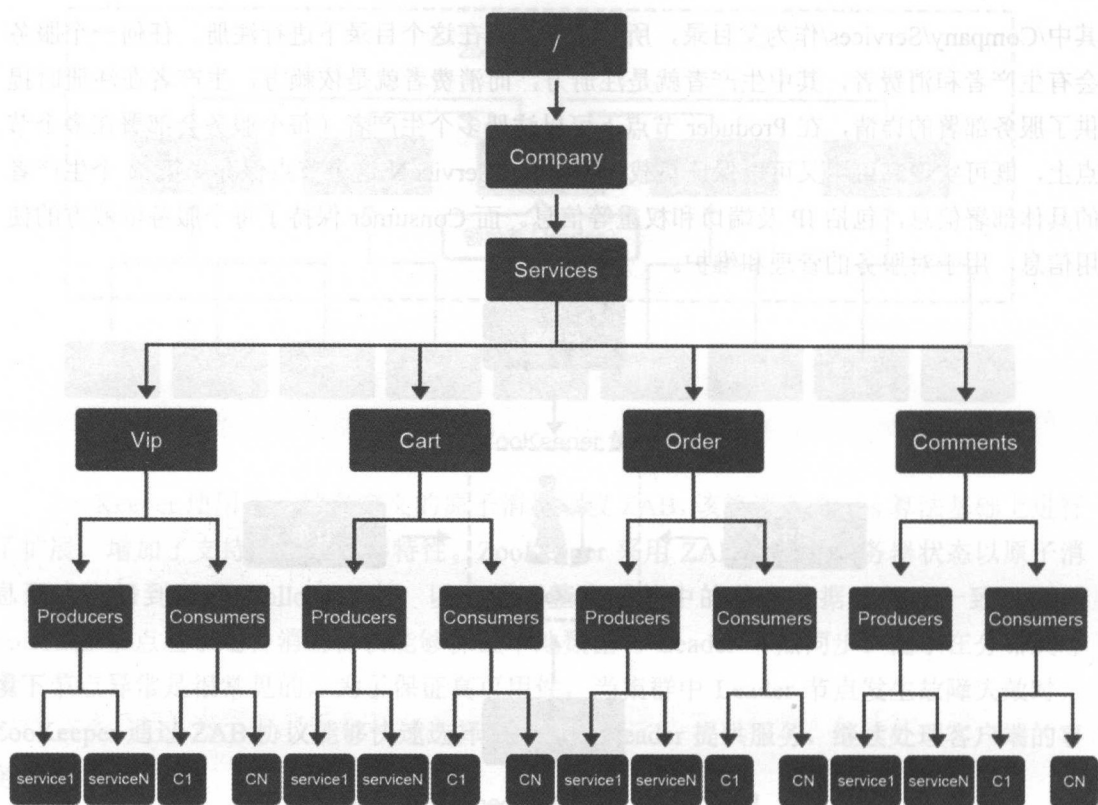


图 5-16 基于 ZooKeeper 的服务发现流程

(2) Etcd

Etcd 简介

Etcd 是由 CoreOS 公司发起的开源的分布式可靠键值存储系统。它使用 Raft 一致性算法保证 Etcd 的高可用性及强一致性。多年发展后，作为 CoreOS 核心组件和 Kubernetes、Cloud Foundry、Mailgun、Apache Mesos 及 Mesosphere Datacenter OS 等众多著名开源项目的支撑，Etcd 本身的可靠性足以支撑生产环境应用。此外，Etcd 本身也比传统的 ZooKeeper 更加轻量，并且文档完成度也比较高。因此，Etcd 也是比较适合作为服务发现系统的支撑组件。

核心能力

1. Etcd 构成集群后可以保证高可用性和强一致性。它用于保证上述特性的算法为更容易理解的一致性算法 Raft，该算法由斯坦福大学的 Diego Ongaro 和 John Ousterhout 提出，可以认为是经典可靠的一致性算法 Paxos 的等价算法，并且在原论文中已证明 Raft 在结果上等价于（多重）Paxos。由于发表 Raft 算法的论文中提供了许多有利于实现的指引，它的实际实现也在预期上比 Paxos 更不容易出错。

2. Etcd 为了方便不同语言编写的程序作为客户端的使用，提供了 HTTP RESTful API 支持，降低了各种语言使用的门槛。目前众多编程语言（Go、Java 等）都已经有了很成熟的 Etcd 客户端开源库支持，使用也比较便利。

3. 作为键-值存储系统，Etcd 的键按类似于 *nix 系统目录放入结构组织，即从根 / 开始的层级结构。键本身在 RESTful API 中作为 path 的一部分使用简单又直接明了。虽然值也只有字符串的支持，但现在有了如 JSON 等众多的序列化方案，即使只支持字符串值也并不是什么问题。

4. 我们可以看到，如 Kubernetes 等项目是完全依赖 Etcd 作为自己的数据存取方式，并且在生产环境中也已经有了验证，所以对 Etcd 感兴趣的读者也不必担心生产环境的健壮型问题。

集群架构

在配置 Etcd 集群时，集群的节点数最好为奇数，因为偶数个节点的容错能力与比它小 1 的奇数节点容错能力相同。官方文档推荐的节点数目为 3、5 或 7，并且 7 个节点已经可以在绝大多数情况下提供足够的容错能力。集群搭建时 Etcd 节点的互相发现有 3 种形式：静态、通过 Etcd 自发现（Etcd discovery）及通过 DNS 发现（DNS discovery）。其中，静态为直接指定各 Etcd 节点的 peer url 以完成集群建立。通过 Etcd 自发现有两种方式：一种为通过自有的已经建立的 Etcd 集群相互发现；另一种是通过 Etcd 官方网站提供的服务进行发现，此时需要有到外网的连接能力。通过 DNS 发现则是利用了 SRV 记录。每种方式的具体配置方式请参阅官方文档（<https://coreos.com/etcd/docs/latest/op-guide/clustering.html>）。

Etcd 集群建立后和已有的业务容器整体架构大致如图 5-17 所示（当然 Etcd 本身可以与容器所在节点独立）。



图 5-17 基于 Etcd 的服务发现架构

Etcd 本身的调用接口也是非常简单的，即基于 HTTP 的 REST 风格 API，并且支持 TLS 加密，详细可以参考 Etcd 的官方文档。下面简单描述一个最基本的通过 Etcd 进行服务发现的例子。简单将应用分为前端（frontend）和后端（backend），Etcd 节点的 client url 为 `http://10.0.0.1:2379`。那么典型的场景如下（用 curl 表示请求内容）。在 10.0.1.1 的后端节点 1 将自己注册到 Etcd 上。

```
$ curl -L http://10.0.0.1:2379/v2/keys/backends/be1 -XPUT -d
value=10.0.1.1
{"action": "set", "node": {"key": "/backends/be1", "value": "10.0.1.1", "modifiedIndex": 4, "createdIndex": 4}}
```

在 10.0.1.2 的后端节点 2 将自己注册到 Etcd 上，带 30s 过期时间。

```
$ curl -L http://10.0.0.1:2379/v2/keys/backends/be2 -XPUT -d
value=10.0.1.2 -d ttl=30
{"action": "set", "node": {"key": "/backends/be2", "value": "10.0.1.2", "expiration": "2016-11-10T08:25:53.429081688Z", "ttl": 30, "modifiedIndex": 5, "createdIndex": 5}}
```

30 s 内，前端可以通过 Etcd 获得两个后端节点的 IP。

```
$ curl -L http://10.0.0.1:2379/v2/keys/backends/
{"action": "get", "node": {"key": "/backends", "dir": true, "nodes": [{"key": "be1", "value": "10.0.1.1", "modifiedIndex": 4, "createdIndex": 4}, {"key": "be2", "value": "10.0.1.2", "expiration": "2016-11-10T08:25:53.429081688Z", "ttl": 30, "modifiedIndex": 5, "createdIndex": 5}]}}
```

```
/backends/be1", "value": "10.0.1.1", "modifiedIndex": 4, "createdIndex": 4}, {"key":
"/backends/be2", "value": "10.0.1.2", "expiration": "2016-11-10T08:25:53.42908168
8Z", "ttl": 14, "modifiedIndex": 5, "createdIndex": 5}], "modifiedIndex": 4, "createdI
ndex": 4}}
```

前端可以将接受到的用户请求处理后转发到上一步通过 Etcd 获得的后端 IP。同时，前端可以监听 Etcd 的变化，以在后端的集群发生变化时能够及时获得最新后端配置，比如上述例子，等待过期。

```
$ curl -L 'http://10.0.0.1:2379/v2/keys/backends?wait=true&recursive=true'
{"action": "expire", "node": {"key": "/backends/be2", "modifiedIndex": 6, "cr
eatedIndex": 5}, "prevNode": {"key": "/backends/be2", "value": "10.0.1.2", "expirati
on": "2016-11-10T08:27:52.882626648Z", "modifiedIndex": 6, "createdIndex": 6}}
```

上述过程发生时，请求和回复有一定的时间差，并不是立刻回复的，因为是属于监听变化而不是单纯的获取。过期后，再次获取后端节点 IP，就会剩下节点 1 的 IP。

```
$ curl -L http://10.0.0.1:2379/v2/keys/backends/
{"action": "get", "node": {"key": "/backends", "dir": true, "nodes": [{"key": "
/backends/be1", "value": "10.0.1.1", "modifiedInd
ex": 3, "createdIndex": 3}], "modifiedIndex": 4, "createdIndex": 4}}
```

当然，真正的应用场景不大可能通过 curl 命令行形式完成服务发现。不过可以看到，Etcd 对请求的返回结果是规整的 json 格式，同时 Etcd API 也都是 HTTP 的形式，所以各种编程语言已有的 Etcd 客户端也比较多，可以直接拿来使用。并且，上述场景只是一个最简单的服务发现流程，很多如失效时间等功能并没有使用，具体使用方式可以参照 Etcd 的官方文档及各编程语言对应的 Etcd client 包文档，相信读者可以用 Etcd 组合方式做出符合自己需要的服务发现。

另外，如果不想对已有实现产生过多侵入，也可以考虑采用 registrator 和 confd 作为已有 Docker 封装的服务接入 Etcd 服务发现的方式。其中，registrator 支持通过 Docker 的 Socket 监听 Docker 事件，从而将所有新增的运行中容器按照它们在启动时定义的名字（即 Docker 命令行中 --name 参数指定的名字）；confd 可以监听 Etcd 指定的目录和自定义配置生成模板，从而将 Etcd 中存储的值转化成自定义形式的配置文件。具体使用方式可以参照两者的官方文档。使用后架构如图 5-18 所示。

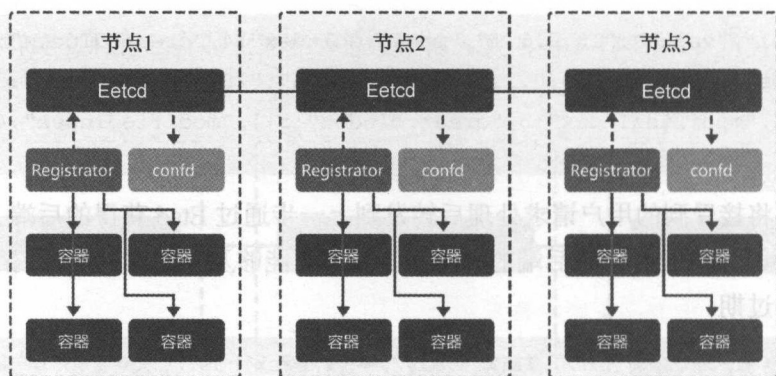


图 5-18 采用自定义配置文件的 Etcd 服务发现架构

综上所述，以 Etcd 为中心的服务发现架构，可以使用比 ZooKeeper 更轻便但又有同样强一致性和高可用性的 Etcd，确保服务发现所用关键数据的正确性，同时支持超时的功能使得过期数据自动失效，再加上方便的 API，使得支撑项目变得十分容易。当然，Etcd 集群配置依然会有些复杂，此外社区开发太活跃可能引入的不稳定性或缺陷也需要权衡，相比之下，ZooKeeper 已经过多年生产环境的洗礼，其稳定性相对更高。以上都是真正在生产环境中使用所需要考虑的。

(3) Consul

Consul 简介

Consul 是 HashiCorp 公司推出的一款服务发现与服务配置工具，基于 Raft 协议，采用 Go 语言实现，主要提供以下功能。

- 服务发现：Consul 客户端与服务提供者部署在同一机器上，集群内的其他 Consul 客户端可通过 HTTP 或 DNS 方式发现此服务。
- 健康检查：Consul 内置健康检查功能，Consul 客户端可配置健康检查，其结果用于决定服务是否可继续对外提供服务。
- key/value 存储：Consul 提供 key/value 存储功能，并提供简单易理解的 HTTP API 用于操作 key/value 数据。
- 多数据中心：Consul 天然支持多数据中心。

核心能力

如官网所述, Consul 最核心的能力是提供服务发现功能, 并自带健康检查功能, 提供 key/value 存储。类似的服务发现产品有 ZooKeeper 与 Etcd, Consul 与它们相比具有以下优势。

- 算法简单易理解。Consul 与 Etcd 均采用 Raft 算法来保证一致性, 算法简单易理解, 而 ZooKeeper 采用的是复杂的 Paxos。
- 支持多数据中心。Consul 支持多数据中心, 而 ZooKeeper 与 Etcd 均不支持。
- 内置健康检查功能。Consul 支持健康检查功能, 而 Etcd 不支持。
- 支持多种服务查询方式。Consul 支持 HTTP 与 DNS 两种协议的服务查询, 而 Etcd 仅支持 HTTP 协议查询, ZooKeeper 集成太复杂。
- 官方提供 Web 管理界面。Consul 官方提供 Web 管理界面, 用于集群 service、node、key/value、ACL 等管理, 而 Etcd 无此功能。

另外, Consul 提供的健康检查功能非常丰富, 分为系统级和应用级。系统级的健康检查用于监控 node 的健康状态; 应用级健康检查与 service 紧密相关, 监控 service 的健康情况。健康检查可预先在配置文件中定义, 也可在运行状态动态通过 HTTP 接口加入。健康检查类型有以下 5 种。

- Script + Interval: 这种健康检查须依赖外部应用进行, script 字段指明脚本位置, interval 指定触发的调用间隔。
- HTTP + Interval: 这种检查采用定时发送 HTTP GET 请求至指定的 URL, 健康状态依赖于响应码, 2xx 表示正常, 429 表示请求过多, 只是警告, 其他的响应码表示应用已经宕机。
- TCP + Interval: 这种检查通过定时发送 TCP 连接至指定的 IP/hostname 与 port, 若 hostname 没有指定, 默认为 localhost。服务的状态依赖于 TCP 连接是否成功, 若成功建立连接, 健康状态为 success, 否则状态为 critical。
- Time to Live (TTL): 通过 TTL 字段指定时间, 健康状态通过 HTTP 接口阶段性更新, 若在指定的 TTL 内更新失败, 表示服务已宕机。
- Docker + Interval: 这种检查依赖部署在 Docker 容器中的应用, 通过 Docker Exec

API 触发运行在容器中的应用进行检查,要求 Consul agent 用户有权限访问 Docker HTTP API 或 UNIX Socket。

集群架构

通常,生产环境集群需要部署多个 server,以保证任何时候都有 $(N/2)+1$ 个 server 可用,否则会导致整个集群不可用。添加 server 时,最好逐个添加,等新加的 server 数据同步完成后,再添加下一个,否则会有数据丢失的风险。Consul 架构如图 5-19 所示。

数据中心 1

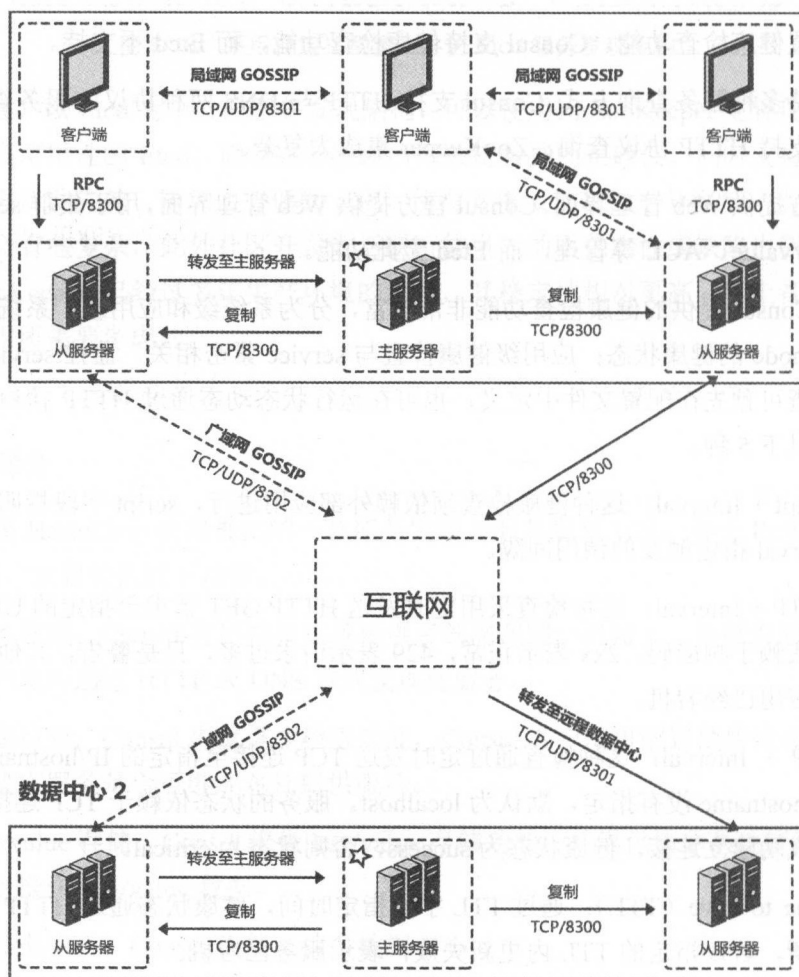


图 5-19 Consul 集群架构

一个 datacenter 内的 node 节点间通过 gossip 协议通信。采用 gossip 协议目的有以下 4 点。

1. 无须在 client 中配置 server 的 IP，自动进行服务发现。
2. node 节点故障检查服务不用部署在 server 上，它支持分布式部署，这样故障检查服务可以拥有更好的水平扩展性。
3. 当有重要事件发生时，可进行消息通知。
4. 多 datacenter 间的 server 通信通过 WAN gossip 进行，主要用于跨 datacenter 发现其他 datacenter 的 node 节点。

Consul 集群实践

1. 获取路径

官方地址是 <https://www.consul.io/>，源码地址是 <https://github.com/hashicorp/consul>，下载地址（含 UI）是 <https://www.consul.io/downloads.html>。

2. 环境准备

| hostname | IP | role | DataCenter | OS |
|--------------------------------|----------------|--------|------------|---------|
| qa-control-consul-test-server | 10.180.148.239 | Server | dc1 | Debian7 |
| qa-control-consul-test-client1 | 10.180.148.240 | Client | dc1 | Debian7 |
| qa-control-consul-test-client2 | 10.180.148.242 | Client | dc1 | Debian7 |

在以上各机器上分别下载 Consul 压缩包，下载方式如下。

```
wget
https://releases.hashicorp.com/consul/0.7.0/consul_0.7.0_linux_amd64.zip
```

另外在 server 这台机器上下载 Consul UI，下载方式如下。

```
wget
https://releases.hashicorp.com/consul/0.7.0/consul_0.7.0_web_ui.zip
```

注意：Consul UI 可安装在集群中的任一台 node，不仅限于 server。

3. Consul 安装

解压 Consul 安装包，将解压出来的 Consul 可执行文件放置到 /usr/local/bin 目录（其他

目录亦可,但必须保证最终将对应路径添加至 PATH)。接着,可在终端下输入 `consul version` 验证安装是否完成。

本次实践,Consul UI 搭建在 server 这台 node 上,解压 Consul UI 压缩包至 `/root/consul-web/`。

4. Consul 集群搭建

假设 server、client1、client2 上的服务分别监听在 7070、8080、9090 端口上,也可通过 `python -m SimpleHTTPServer 7070 &` 进行简单模拟。

在各 node 上创建目录 `/etc/consul.d/` (目录名称可为其他) 用于存储 Consul 配置,配置包含系统配置与应用服务配置,目前仅支持 json 格式,文件以 .json 结尾。server 与 client 配置分别如下。

● server

系统级配置 (default.json)

```
{
  "server": true,
  "bootstrap": true,
  "node_name": "consul-server",
  "client_addr": "0.0.0.0",
  "bind_addr": "10.180.148.239",
  "data_dir": "/data",
  "ui_dir": "/root/consul-web"
}
```

其中,server 为 true 表示 consul agent 以 server 模式运行,node_name 用于稍后的服务发现,bind_addr 配置的 IP 用于集群内部通信,与集群中其他节点可互联。默认为 0.0.0.0,这时 Consul 将使用第一个有效的私有 IPv4 地址。data_dir 存放 agent 状态信息,ui_dir 值为 Consul UI 存放路径,client_addr 为绑定到 client 接口的地址,可通过 HTTP、DNS、RPC 访问,默认值为 127.0.0.1,只能在本机访问。

应用级配置 (web.json)

```
{
  "service": {
```

```
"name": "web",
"tags": ["web"],
"address": "10.180.148.239",
"port": 7070,
"check": {
  "id": "web-check",
  "name": "Web health check on port 7070",
  "http": "http://localhost:7070",
  "interval": "5s",
  "timeout": "1s"
}
}
```

● client1

系统级配置 (default.json)

```
{
  "node_name": "consul-client1",
  "bind_addr": "10.180.148.240",
  "client_addr": "0.0.0.0",
  "data_dir": "/data",
  "start_join": ["10.180.148.239"]
}
```

应用级配置 (container.json)

```
{
  "service": {
    "name": "container",
    "tags": ["container"],
    "address": "10.180.148.240",
    "port": 8080,
    "check": {
      "http": "http://localhost:8080",
      "interval": "5s",
```



```
    "timeout": "1s"
  }
}
```

● client2

系统级配置 (default.json)

```
{
  "node_name": "consul-client2",
  "bind_addr": "10.180.148.242",
  "client_addr": "0.0.0.0",
  "data_dir": "/data",
  "start_join": ["10.180.148.239"]
}
```

应用级配置 (controller.json)

```
{
  "service": {
    "name": "controller",
    "tags": ["controller"],
    "address": "10.180.148.242",
    "port": 9090,
    "check": {
      "http": "http://localhost:9090",
      "interval": "5s",
      "timeout": "1s"
    }
  }
}
```

各个节点上的配置文件添加后，然后分别执行以下步骤。

1. 在 server 上执行 `consul agent -config-dir=/etc/consul.d &`，启动过程中，控制台的日志如图 5-20 所示。

```

root@qa-control-consul-test-server:~# consul agent -config-dir=/etc/consul.d &
[1] 13492
root@qa-control-consul-test-server:~# ==> WARNING: Bootstrap mode enabled! Do not enable unless necessary
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
    Version: 'v0.7.0'
    Node name: 'consul-server'
    Datacenter: 'dc1'
    Server: true (bootstrap: true)
    Client Addr: 0.0.0.0 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC: 8400)
    Cluster Addr: 10.180.148.239 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
    Atlas: <disabled>

==> Log data will now stream in as it occurs:

2016/10/27 19:31:01 [INFO] raft: Restored from snapshot 9-65621-1477555050336
2016/10/27 19:31:01 [INFO] raft: Initial configuration (index=1): [{Suffrage:Voter ID:10.180.148.239:8300 Address:10.180.148.239:8300}]
2016/10/27 19:31:01 [INFO] raft: Node at 10.180.148.239:8300 [Follower] entering Follower state (Leader: "")
2016/10/27 19:31:01 [INFO] serf: EventMemberJoin: consul-server 10.180.148.239

```

图 5-20 服务端启动过程中的日志

2. 在 client1 与 client2 上分别执行 `consul agent -config-dir=/etc/consul.d &`，客户端启动过程中，控制台的日志如图 5-21、图 5-22 所示。

server 与 client 都启动完成后，可通过 `consul members` 在任一台机器上查看集群成员信息，更为详细的信息，可添加 `detailed` 参数。`consul members` 的查看方式是基于 gossip protocol 这种最终一致性协议进行的，某一时刻，可能本地的 agent 上看到的内容与 server 不一致。不过也可采用 HTTP API 进行强一致性方式查看，具体操作为：

```
curl localhost:8500/v1/catalog/nodes 或 curl node-ip:8500/v1/catalog/nodes.
```

```

root@qa-control-consul-test-client1:~# ==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
    Join completed. Synced with 1 initial agents
==> Consul agent running!
    Version: 'v0.7.0'
    Node name: 'consul-client1'
    Datacenter: 'dc1'
    Server: false (bootstrap: false)
    Client Addr: 127.0.0.1 (HTTP: 8500, HTTPS: -1, DNS: 8600, RPC: 8400)
    Cluster Addr: 10.180.148.240 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
    Atlas: <disabled>

==> Log data will now stream in as it occurs:

2016/10/27 19:33:12 [INFO] serf: EventMemberJoin: consul-client1 10.180.148.240
2016/10/27 19:33:12 [INFO] agent: (LAN) joining: [10.180.148.239]
2016/10/27 19:33:12 [INFO] serf: EventMemberJoin: consul-server 10.180.148.239
2016/10/27 19:33:12 [INFO] agent: (LAN) joined: 1 Err: <nil>
2016/10/27 19:33:12 [INFO] consul: adding server consul-server (Addr: tcp/10.180.148.239:8300) (DC: dc1)
2016/10/27 19:33:12 [INFO] agent: Synced service 'container'

```

图 5-21 客户端 (client1) 启动过程中的日志


```

root@qa-control-consul-test-client2:~# dig @10.180.148.239 -p 8600 consul-server.node.consul
; <<> DiG 9.8.4-rpz2+r1005.12-P1 <<> @10.180.148.239 -p 8600 consul-server.node.consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60631
;; flags: qr aa rd: QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;consul-server.node.consul.      IN      A

;; ANSWER SECTION:
consul-server.node.consul. 0      IN      A      10.180.148.239

;; Query time: 1 msec
;; SERVER: 10.180.148.239#8600(10.180.148.239)
;; WHEN: Thu Oct 27 20:21:57 2016
;; MSG SIZE rcvd: 59

```

图 5-24 DNS 查看方式

node 查询的 DNS 名称为 consul-server.node.consul，其中，consul-server 为 node 名称，node 子域表示我们要查询的为 node 资源，如要查询 service 资源，则子域为 service。顶级域 Consul 为默认值，不过也可通过 domain 参数进行配置。

6. Consul UI

Consul 提供了更为直观的 Web 管理界面，如图 5-25 所示，用户可通过 Web UI 查看 service、node、key/value、ACL 等信息。



图 5-25 Consul 管理界面

5.5.2 服务治理

随着业务逐渐复杂，在微服务架构下，服务数量会越来越多，引入的问题也会越来越复杂。如何在业务发展的同时保障服务的 SLA 和最大化利用机器资源是摆在技术人员面前一个很大的挑战。这时就需要一个统一的服务治理机制对所有服务进行统一管控，保障服务正常运行。

服务治理范围覆盖了服务的整个生命周期，从服务建模开始，到开发、测试、审批、发布、运行时管理，以及最后的下线。我们通常说的服务治理主要是指服务运行时的治理，一个好的服务治理框架要遵循“在线治理，实时生效”原则，只有这样才能真正保障服务整体质量。下面介绍服务治理策略在服务运行时的应用。

1. 服务越来越多，配置项越来越多，利用统一注册中心解决服务发现和配置管理问题。
2. 服务之间存在多级依赖，靠人工已经无法理清，还要避免潜在的循环依赖问题，我们需要依赖管理机制，支持导出依赖关系图。
3. 服务的性能数据和健康状态数据是服务治理的重要依据，比如访问量、响应时间、并发数等，因此需要有监控、健康检查和统计服务。
4. 当一个服务的访问量越来越大，需要对服务进行扩容，然后在客户端进行流量引导和优先级调度。
5. 面对突发流量，已经无法通过扩容解决问题时，要启用流量控制，甚至服务降级。
6. 随着业务持续发展，要提前进行容量规划，结合服务监控数据，以确认当前系统容量能否支撑更高水位的压力。
7. 等越来越多的微服务上线之后，从安全角度看，我们需要实施明确的权限控制策略和服务上下线流程。
8. 通过一系列的服务治理策略，最终通过数据证明系统对外承诺的 SLA。

典型治理方式

弹性扩缩容

4.2 节提到基于负载均衡的应用弹性伸缩方案，只要将应用系统设计成无状态，在需要伸缩的时候修改负载均衡代理配置，就可以方便地水平扩容应用系统，提高系统承载

能力。

在云原生应用架构里，技术人员其实有更多的选择。对于无状态服务，配合云平台提供的 AutoScaling 能力，能够快速弹性扩容，实施 DevOps。在这里，弹性扩缩容是一种重要的服务治理手段。下面以网易云为例介绍基于 Kubernetes 的 AutoScaling 机制实现弹性伸缩。

弹性扩缩容的实现

网易云采用 Kubernetes 实现应用管理，而 Kubernetes 的 Horizontal Pod Autoscaler (HPA) 组件专门设计用于应用弹性扩容的控制器，它通过定期轮询 Pod 的状态（CPU、内存、磁盘、网络，或者自定义的应用指标），当 Pod 的状态连续达到提前设置的阈值时，就会触发副本控制器，修改其应用副本数量，使得 Pod 的负载重新回归到正常范围之内。如图 5-26 所示。

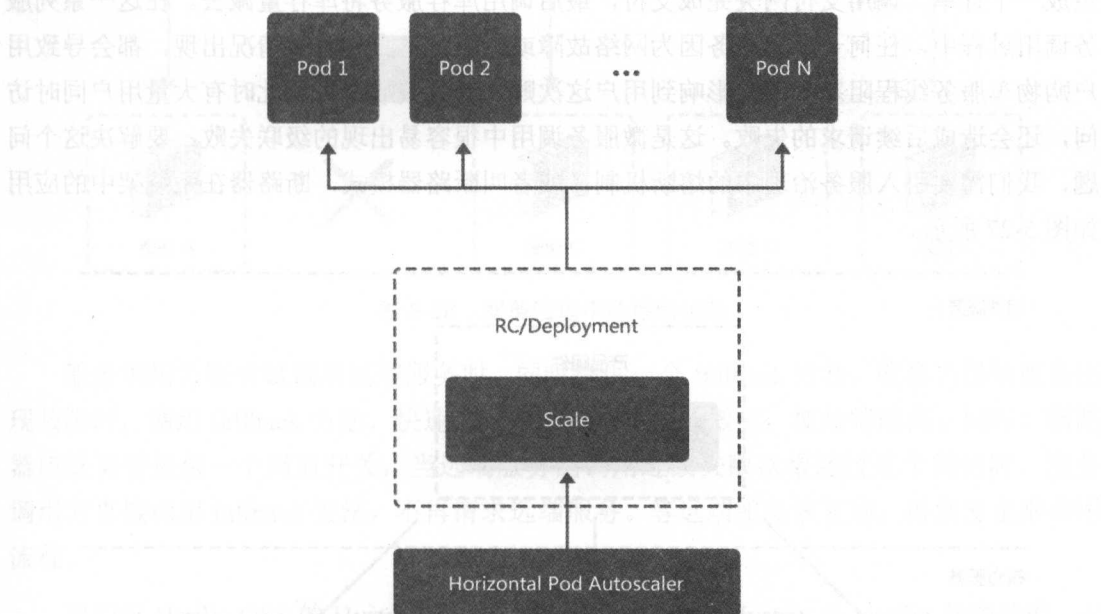


图 5-26 基于 Kubernetes 的弹性扩缩容

例如促销活动服务的应用层是一个无状态应用，当前有两个副本，我们把弹性扩容的 CPU 使用率阈值设置为 50%。但是促销当天涌入的流量远远超过预期，使得两个副本的 CPU 使用率分别达到了 80%以上，HPA 控制器监控到这种变化，于是通知副本控制器将促

销活动服务的副本数量升到 4 个。当流量峰值过后, 4 个副本的 CPU 使用率慢慢降到 10% 以下, HPA 控制器计算得出两个副本即可满足负载要求, 于是通知副本控制器将应用副本数量变为 2。

HPA 控制器的副本伸缩算法可以参考 Kubernetes 文档。

熔断机制

微服务架构中, 各服务通过服务发现的方式互相依赖, 虽然从单个服务看来能获得非常好的隔离性, 不会因为某个进程或者服务宕掉对其他服务造成直接影响, 但是从业务角度来看, 单个服务实例故障还是可能造成业务访问出现问题, 轻则影响服务调用方出现延迟和负载上升, 重则造成业务整体异常。

比如, 一个简单的电商场景, 用户通过网站下单购买一件商品, 首先将调用订单服务生成一个订单, 调用支付网关完成支付, 最后调用库存服务将库存量减去。在这一系列服务调用过程中, 任何一个子服务因为网络故障或者服务本身异常等情况出现, 都会导致用户购物车服务线程阻塞, 不仅影响到用户这次购物行为失败, 如果此时有大量用户同时访问, 还会造成后续请求的失败。这是微服务调用中很容易出现的级联失败, 要解决这个问题, 我们需要引入服务治理中的熔断机制, 或者叫断路器模式, 断路器在系统架构中的应用如图 5-27 所示。

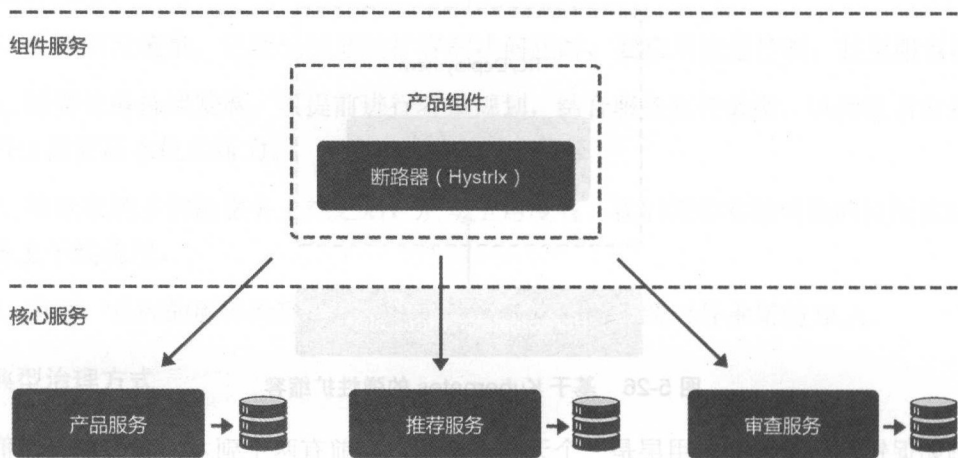


图 5-27 断路器在系统架构中的应用

断路器是一个开关，本意是指电路系统上的一种保护线路电流过载的一种装置，当线路中电流太大或者发生短路时，断路器开关打开，电路切断，防止引起更加严重的后果。引申到微服务治理策略中，断路器的作用就是避免故障或者异常在微服务调用链中蔓延。它的工作机制如图 5-28 所示。

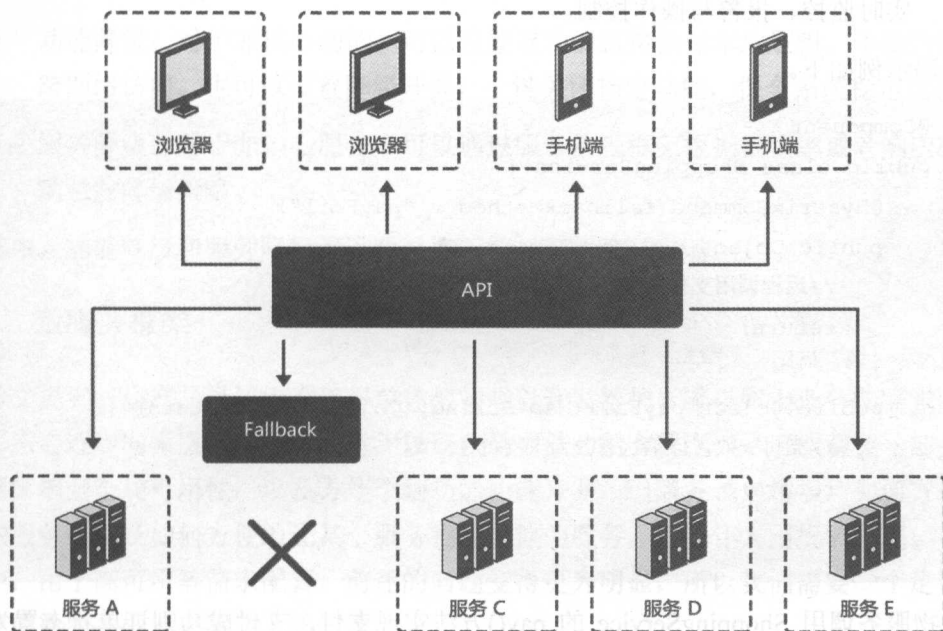


图 5-28 服务治理中的熔断机制

服务调用方在尝试调用远端服务时，同时提供一个 fallback 方法，就是当远端服务出现故障时，调用 fallback 方法，快速返回结果，避免级联效应，使故障隔离。同时，断路器应该需要提供一个阈值开关，当远端服务的调用连续失败次数超过某个阈值时，服务调用方直接调用 fallback 方法，不再请求远端服务。等远端服务恢复后，再恢复正常调用流程。

我们以 Netflix OSS 的 Hystrix 为例说明断路器的实现。Hystrix 是 Netflix 开源的库，主要提供分布式服务间交互的延时容忍与容错机制，隔离了服务间的访问入口，防止整个链路上某服务调用不通导致系统雪崩，提供 fallback（降级）机制以便增强系统弹性。另外，还提供了服务治理与监控功能。

Hystrix 主要提供以下几个功能。

- 为服务提供保护，控制延迟和故障。
- 避免复杂系统的级联故障。
- 快速失败与高效恢复。
- 实时监控、报警与操作控制。

代码示例如下。

```
@Component
public class ShoppingService {
    @HystrixCommand(fallbackMethod = "payFail")
    public Object pay(Map<String, Object> parameters) {
        //远程调用支付服务
        return;
    }
    public Object payFail(Map<String, Object> parameters) {
        //支付失败，订单状态改为未支付
        return;
    }
}
```

购物服务调用 `ShoppingService` 的 `pay()` 方法实现支付，支付成功则订单状态置为待发货，若支付过程中支付网关服务出现异常，导致 `pay()` 方法调用失败，`ShoppingService` 的断路器会调用 `payFail()` 方法实现失败处理，将订单状态改为未支付状态，后续用户可以通过界面选择订单继续支付。如果支付网关服务较长时间无法恢复，当 `pay()` 连续失败次数超过阈值，熔断机制开启，断路器打开，每次对 `ShoppingService` 的 `pay()` 方法的调用退化为对 `payFail()` 方法的调用，直至支付网关服务恢复正常。熔断机制在服务治理中的作用主要体现在对故障的隔离上，避免调用出现链式雪崩。

服务降级

服务降级也是服务治理策略中重要的一环。当业务出现流量峰值，或者系统中某个组成部分出现故障，保证系统整体功能仍然可用，我们可能需要停掉一些不太重要的周边系统，从而保证核心服务的 SLA。比如电商系统在进行大促时，往往会弃车保帅，优先选择停止“猜你喜欢”、“评论”等不重要的系统，保障购物车、支付系统可用。在微服务架构

里，每个服务无论是服务提供方还是服务调用方，都应该围绕 SLA 制定不同的降级策略。按降级粒度粗细我们可以制定接口降级、功能降级、服务降级。

- 接口降级：对于非核心接口，设置为直接返回空或异常，可以在高峰期有效减少接口逻辑对资源（CPU、内存、网络 I/O、磁盘 I/O 等）的占用和消耗。
- 功能降级：对于非核心功能，可以设置该功能直接执行本地逻辑，不做跨服务、跨网络访问。也可以设置降级开关，一键关闭指定功能，保全系统稳定运行。
- 服务降级：对于非核心服务，可以通过服务治理框架根据错误率或者响应时间自动触发降级策略。

其中，功能降级和服务降级可以通过熔断机制和断路器实现。

5.5.3 微服务框架

前文提到，当产品足够庞大使得单体架构难以开发维护、难以跟上业务的需求时，产品的形态会逐步向微服务的方向演进，以期获得更低的耦合程度，尽可能避免一部分功能的问题影响到全局可用性，以及方便个别功能快速开发。但服务越来越多，如何方便地管理这些服务，以及如何方便地让某个服务找到其他的服务，成了让人困扰的问题。在生产环境中，由于高可用等需求配置，前述的问题变得更为明显，所以我们需要一个足够健壮又方便的微服务框架来解决上述问题。

1. Spring Cloud

简介

Spring Cloud 是 Pivotal 推出的基于 Spring Boot 的一系列框架的集合，旨在帮助开发者快速搭建一个分布式的服务或应用。它由众多子项目组成，如 Spring Cloud Config、Spring Cloud Netflix、Spring Cloud Consul 等，提供了搭建分布式系统及微服务常用的工具，如配置管理、服务发现、断路器、智能路由、微代理、控制总线、一次性 token、全局锁、选主、分布式会话和集群状态等。

Spring Cloud 基于 Spring Boot，这也意味着，其使用方式如 Spring Boot 简单易用，极大简化了分布式系统基础设施的开发，其自带默认配置可以快速搭建框架，如果需要可以通过配置及扩展实现定制化配置。由上文可以看到 Spring Cloud 提供了众多的工具，但是

它并没有重复造轮子，而是选用目前各家公司开发的比较成熟的、经得住实践考验的服务框架，通过 Spring Boot 进行封装集成并简化其使用方式，如 Spring Cloud Netflix 整合了 Netflix OSS，可以直接在项目中使用 Netflix 组件。

使用 Spring Cloud 搭建的服务或应用，可以适用于任何分布式环境，包括开发者自己的计算机笔记本、Docker 或者其他专业的云计算平台（如 Cloud Foundry、网易云基础服务平台等）。

核心能力

Spring Cloud 是一套完整的分布式系统解决方案，它的子项目涵盖了所有分布式系统所需要的基础设施。而且作为 Spring 的项目，能够与 Spring Framework、Spring Boot、Spring Data 等其他 Spring 项目完美融合，极大减少了已有项目的迁移成本。

Spring Cloud 致力于为典型的使用场景提供好的开箱即用体验，同时提供扩展机制来满足其他需求。

- 分布式的/版本化配置
- 服务注册和发现
- 路由
- 服务间调用
- 负载均衡
- 断路器
- 全局锁
- 选主、集群状态
- 分布式消息机制

Spring Cloud 很多特性的使用都是采用很直白的方式（如一个 classpath 的改变或一个注解）。服务发现客户端的示例。

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {
```

```
public static void main(String[] args) {  
    SpringApplication.run(Application.class, args);  
}
```

应用场景

Spring Cloud 可以适用于各种分布式系统服务或应用的场景,如近两年非常火的微服务架构。其满足了构建微服务所需的所有解决方案。

- 使用 Spring Cloud Config 可以实现统一配置中心,对配置进行统一管理
- 使用 Spring Cloud Netflix 可以实现 Netflix 组件的功能。如服务发现 (Eureka)、智能路由 (Zuul)、客户端负载均衡 (Ribbon) 等。

■ Eureka 是 Netflix 开源的服务发现框架,提供中间层服务的负载均衡和故障转移功能。在 Netflix 内部, Eureka 除了用于中间层负载均衡外,还用于以下几个方面。

1. 协助 Netflix Asgard (让云部署更便捷的开源服务) 在出现故障时 100+ 实例的快速回滚。
2. 针对 Cassandra 部署,将实例从流量中导出以便维护。
3. 标识 Memcached 缓存集群中一系列节点。
4. 携带其他应用中有关服务的指定元信息。

■ Zuul 是一个智能网关服务,主要提供动态路由、监控、安全认证等功能。所有来自用户的请求先到达 Zuul,再由它分发。它有以下功能。

1. 安全认证与防爬虫:所有外部请求必须经过网关,网关可以集中对访问进行安全控制。
2. 审查与监控:集中监控访问量、调用延迟、错误计数与访问模式,为后端的性能优化或扩容提供数据支持。
3. 动态路由:按需动态路由请求至不同的后端集群。
4. 压力测试:逐渐增加集群流量以便测量集群性能。

5. 减压 (Load shedding): 为每类请求分配容量, 当到达上限后, 丢弃请求。
6. 静态响应处理: 在入口 (edge) 处构建响应, 而不用 forward 它们至内部集群。

■ Ribbon 是客户端的 IPC (进程间通信) 库, 内置软件负载均衡, 已经过大规模考验。主要提供以下服务。

1. 负载均衡。
2. 容错。
3. 在异步&响应式模型中支持多种协议 (HTTP、TCP、UDP)。
4. 缓存和批处理。

- 如果不想使用 Eureka 实现服务发现等, 可以使用 Spring Cloud Consul 通过 Consul 实现服务发现及配置管理, 或者使用 Spring Cloud ZooKeeper 通过 Apache 的 ZooKeeper 实现服务发现及配置管理。

其他子项目在此就不一一列举了, 建议大家自行查阅 Spring Cloud 的官方文档 (<http://projects.spring.io/spring-cloud>), 以对 Spring Cloud 有更全面的了解。

2. Dubbo

简介

Dubbo 是阿里巴巴公司针对大规模网站应用研发并开源发布的。它是一个能够支撑分布式服务架构及流动计算架构, 具有调度、发现、监控、治理等功能的一站式微服务解决方案。根据官方数据显示, Dubbo 每天为 2,000+ 个服务提供 3,000,000,000+ 次访问量支持 (<http://dubbo.io/>), 可以说 Dubbo 本身足够支持生产环境的大规模访问网站应用。接下来的内容主要参考了 Dubbo 的官方用户指南文档, 为了方便读者能够直观快速地了解 Dubbo 的相关内容, 真正使用时, 还请读者直接参考官方文档以获取最新的信息, 以免使用过时的内容。

核心能力

从图 5-29 可以看到, Dubbo 支持了相当丰富的服务治理能力。在开源版本中, Dubbo 提供了服务注册/发现的能力和在此基础上实现的路由规则、动态配置、服务降级、访问

控制、权重调整和负载均衡等管理能力。上述管理能力有专门的管理运维控制台可以使用，极大地方便了运维。前面提到的服务注册/发现能力对应的 Dubbo 的服务发现体系如图 5-30 所示。

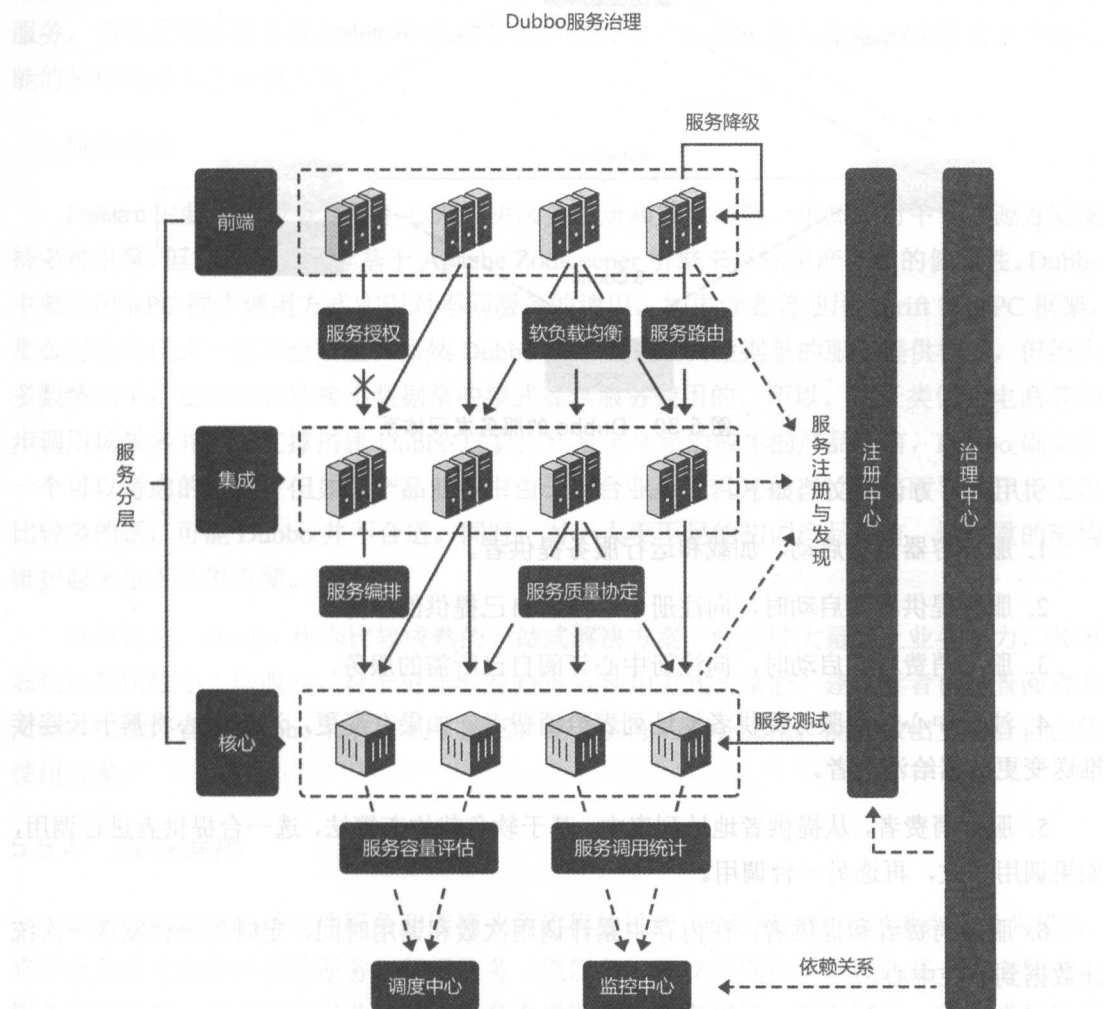


图 5-29 Dubbo 服务治理

Dubbo 架构

---> Init ---> Async —> Sync

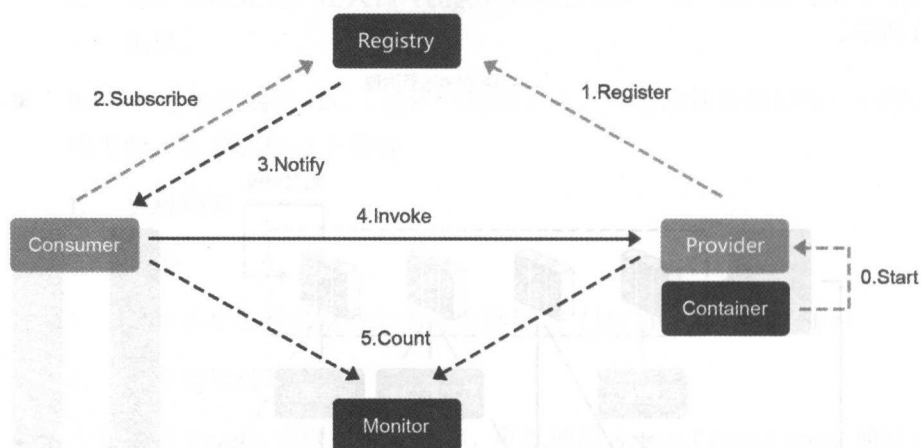


图 5-30 Dubbo 的服务发现体系

引用其官方说明文档如下。

1. 服务容器负责启动、加载和运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

同时，在一次成功获取提供者列表后，服务消费者将缓存这份列表，所以过程中如果出现注册中心不可用的情况，只要列表中的服务提供者全部失效，服务就依然可以正常使用，保证其可用性。注册中心、服务提供/消费方均相互以长连接连通。注册中心通过长连接感知服务提供者存在，如果服务提供者失效，注册中心将推送事件通知消费者更新列表。而注册中心本身也是对等集群，并会缓存服务列表已被数据库失效时继续提供发现功能。

因此，总体上讲，Dubbo 本身的服务发现结构有很强的可用性与健壮性，足够支持高访问量的网站。

另外，注册中心为对等集群，可随时扩大注册中心实例数。如果服务提供方为无状态服务，则动态增减服务提供方对整体服务提供无影响。Dubbo 整体的框架设计对于今后可能的架构升级不会有很大阻力。

应用场景

Dubbo 主要通过服务注册中心来解决服务注册和发现问题。虽然注册中心开源方案支持多种引擎，但官方依然推荐基于 Apache ZooKeeper 引擎来保证生产环境的健壮性。Dubbo 主要采用 RPC 同步调用方式实现对不同服务的调用。如果读者曾使用 Thrift 等 RPC 框架，那么对这种模式一定不会陌生。虽然 Dubbo 支持短连接大数据量的服务提供模式，但绝大多数情况下都是使用长连接小数据量的模式提供服务使用的。所以，对于类似于电商等同步调用场景多并且能支撑搭建 Dubbo 这套比较复杂环境的成本的产品而言，Dubbo 确实是一个可以考虑的选择。但如果产品业务中由于后台业务逻辑复杂、时间长而导致异步逻辑比较多的话，可能 Dubbo 并不合适。同时，对于人手不足的初创产品而言，这么重的架构维护起来也不是很方便。

总而言之，Dubbo 作为比较成熟的一站式解决方案，经历过大量线上业务压力，其可靠性还是比较令人信服的。对于更详细的 Dubbo 使用方式与功能，建议读者自行查阅官方文档。相信读者对 Dubbo 有详细的了解后，结合实际的业务应用，能够做出更适合自己的使用方案。

5.5.4 服务编排

服务编排描述的是软件服务和连接业务流程的过程，并可用自然描述语言记录为模板。它联合企业内部和外部的服务，每项服务（微服务）仅仅实现业务流程的某一特定环节。服务编排工具一般具有定义业务流程中各个微服务的公有属性、私有属性、创建或释放规则、依赖关系、数据流向、配置管理、集群管理等功能。实现服务编排是一个复杂的过程，除了需求编排工具外，还需要自身业务做到以下几个过程：服务拆分、服务独立部署、服务间的通信和服务发现。

本节主要讨论服务编排的实现。

1. 服务编排实现方式

在大型云平台里编排一般分为资源编排和服务编排两类。资源编排定义所需物理资源，如主机实例、CPU、内存、VPC、路由表、磁盘、数据库等资源组合。服务编排以服务为中心定义服务的弹性伸缩、灰度发布、滚动升级、资源配置等功能组合。基于容器的服务编排需要结合 Docker 对容器定义进行组合，其实现过程如下。

对象定义

服务编排最终是对一个个对象做排列、组合等操作，故必须对整个系统进行对象拆分，并定义对象的基本语义和属性。对象定义包含对象元数据、对象依赖、运行时规则（创建、删除、更新）、条件、映射项、输入、输出、权限和函数。不管是资源还是服务都可以抽象为一个对象。编排就是把多个对象按预先设置好的依赖，运行时规则等对象元数据组合成一个模板。部署模板时平台系统通过语法解析器获得资源调度依赖、资源配置、调度模块等信息，最后根据不同的模块执行对应的功能。

语法解析

语法解析是指对所定的模板格式和对象数据进行识别，根据定义的语义执行不同的功能。常用的语法格式如 XML、JSON、YAML 格式。例如，AWS 编排支持 JSON 格式自定的语法。Kubernetes 服务编排支持 YAML 和 JSON 两种语法格式。Docker Compose 的编排支持 YAML 语法。网易云容器编排支持 Docker Compose 基本命令和扩展命令，同样为 YAML 语法。不管是哪一种语法格式都避免不了定义一些与平台相关的扩展命令，所以语法解析器也会在通用语法规则上进行扩展，或者像 AWS 一样，完全自定义一套语法解析。

任务调度

此处的任务调度仅在模板经过语法解析后，根据模板对象定义的依赖做顺序或并行调度对象到子系统。具体的调度算法由各子系统或子模块内部处理。分解后的每个任务调度执行情况显示返回日志中心。

模板管理

模板管理提供对模板的编辑、更新，这些操作都会记录审计日志。对于一个运行的模板实例可执行更新操作，更新后的配置合并到模板实例配置。模板可以保存在本地、模板

仓库或平台对象存储中。

2. 服务编排实践

网易云基础平台服务提供的编排可以简化基础设施或应用的配置，通过模板快速复制基础设施或应用服务，同时可以控制和跟踪用户对基础设施或应用服务所做的更改。这里以网易云用户的网易三拾众筹为例介绍服务编排实践。网易三拾众筹是网易旗下众筹服务平台，目前已成功完成多个项目资金筹款。三拾众筹建站之初就考虑利用容器的特性来完成系统的快速开发、测试、上线和发布。网易云为三拾众筹提供完备底层容器服务，为其快速开发、测试、上线和发布提供保障。

三拾众筹网站方案

如图 5-31 所示，此案例分为前台和后台管理，前后和后台程序都通过微服务方式实现。网站中各个模块可独立升级、部署。接下来根据微服务架构原则来看具体的需求和规划。

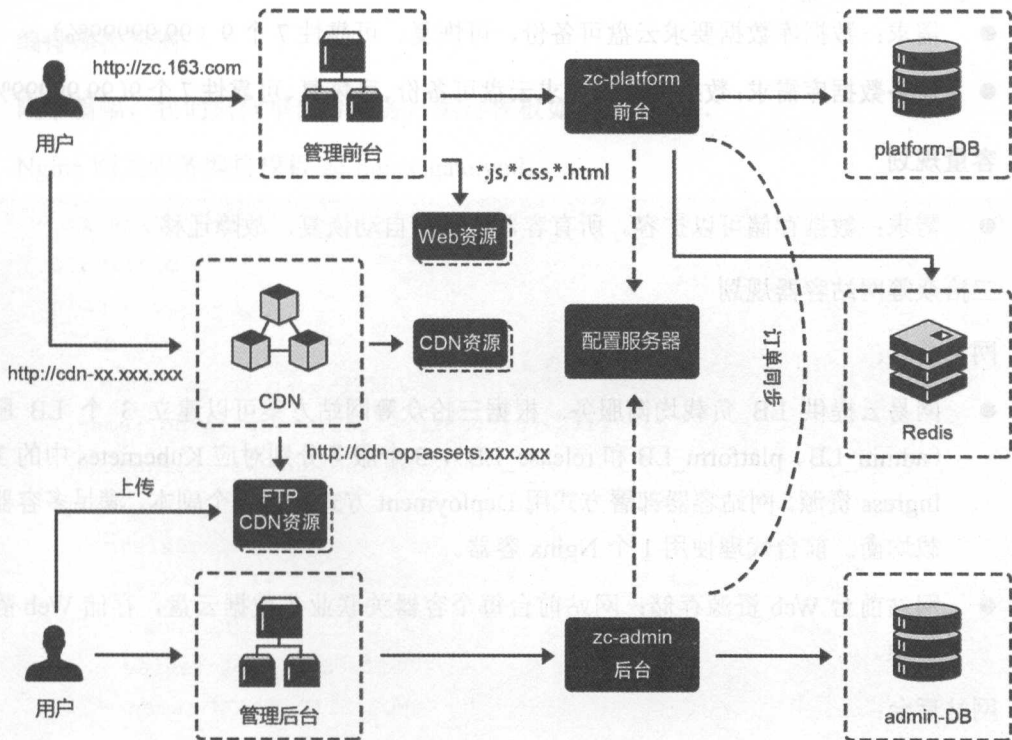


图 5-31 网易三拾众筹网站方案

三拾众筹网站方案需求

- 网站前台需求：业务网站必须容灾、支持负载均衡、前台容器要求按业务量自动伸缩。
- 网站前台 Web 资源存储需求：静态资源存储在云盘、数据不能丢失、可靠性要求 7 个 9 (99.99999%)。
- 网站后台需求：后台要求不影响正常业务、权限控制、监控。

配置管理服务器

- 需求：7*24 小时稳定运行。
- CDN 加速、缓存需求：CDN 服务。

Redis 数据库

- 需求：数据库数据要求云盘可备份、可恢复、可靠性 7 个 9 (99.99999%)。
- 业务数据库需求：数据库数据要求云盘可备份、可恢复、可靠性 7 个 9 (99.99999%)。

容量规划

- 需求：数据存储可以扩容。所有容器要求可自动恢复，故障迁移。

三拾众筹网站容器规划

网站前台：

- 网易云提供 LB 负载均衡服务。根据三拾众筹网站方案可以建立 3 个 LB 服务 (admin_LB、platform_LB 和 release_LB)，3 个服务分别对应 Kubernetes 中的 3 个 Ingress 资源。网站容器部署方式用 Deployment 方式部署 4 个副本，满足多容器负载均衡。前台代理使用 1 个 Nginx 容器。
- 网站前台 Web 资源存储：网站前台每个容器关联业务数据云盘，存储 Web 静态资源。

网站后台：

- Deployment 方式部署 1 个后台代理 Nginx 容器和 1 个后台管理容器。

配置管理服务器:

- Deployment 方式部署 1 个后台管理容器。

CDN 加速、缓存:

- 配置 CDN 服务, 配置对象存储服务。

Redis 数据库:

- Deployment 方式部署 1 个 Redis 容器, 并挂载数据盘。
- 业务数据库配置主备业务服务库 RDS 服务。

综上所述, 三拾众筹网站可以用 10 个容器和对应的 LB、CDN、数据库配置搭建。在编排模板中描述容器的镜像、资源和对平台依赖的数据库、CDN 等信息, 网易云平台可以根据模板为用户部署一套完整的产品。用户通过平台可以方便、快速地对容器进行更新、管理、监控和升级等操作, 更多详情请参考网易云公众号的案例分享。

编排模板示例

限于篇幅, 我们只列举部分服务, 编排模板如下。

Nginx 网关服务编排模板 `zc-nginx-gate.yml`。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: nginx-gate-72551
    name: nginx-gate-72551-3526940187-771i7
    namespace: cw-app-online
spec:
  containers:
  - env:
    - name: JOIN
      value: 10.173.33.115
    - name: NGINX_SITE_CONF
      value: config/wyzc/nginx-gate.conf
    image: hub.c.163.com/comb/cw-cloud-nginx:v1.15.1
```



```

imagePullPolicy: IfNotPresent
name: nginx-gate
resources:
  limits:
    cpu: "1"
    memory: "2109128704"
  requests:
    cpu: "1"
    memory: "2109128704"
securityContext:
  capabilities:
    add:
      - NET_ADMIN
  volumeMounts:
    - mountPath: /app-logs/
      name: log-volume-1-1
dnsPolicy: ClusterFirst
nodeSelector:
  stateful: "false"
restartPolicy: Always
volumes:
  - hostPath:
      path: /container/logs/cw-app-online/nginx-gate/app-logs/
      name: log-volume-1-1

```

众筹前台服务编排模板 `zc-platform.yml`。

```

apiVersion: v1
kind: Pod
metadata:
  labels:
    name: zc-platform-a-72574
    name: zc-platform-a-72574-3305764786-3jwbj
    namespace: cw-app-online
spec:
  containers:

```

```

- env:
  - name: JAVA_OPTS
    value: -Xmx1500m -Xms1500m -Xss256k
  - name: JOIN
    value: 10.173.33.115,10.173.33.128,10.173.33.147
image: hub.c.163.com/comb/cw-zc-platform:v1.13
imagePullPolicy: IfNotPresent
name: zc-platform-a
resources:
  limits:
    cpu: "1"
    memory: "2109128704"
  requests:
    cpu: "1"
    memory: "2109128704"
securityContext:
  capabilities:
    add:
      - NET_ADMIN
volumeMounts:
  - mountPath: /app-logs/
    name: log-volume-1-1
dnsPolicy: ClusterFirst
nodeSelector:
  stateful: "false"
restartPolicy: Always
volumes:
  - hostPath:
      path: /container/logs/cw-app-online/zc-platform-a/app-logs/
      name: log-volume-1-1

```

众筹后台服务编排模板 `zc-admin.yml`。

```

apiVersion: v1
kind: Pod
metadata:

```

```
labels:
  name: zc-admin-a-72581
  name: zc-admin-a-72581-2085285157-erakc
  namespace: cw-app-online
spec:
  containers:
  - env:
    - name: JAVA_OPTS
      value: -Xmx1500m -Xms1500m -Xss256k
    - name: JOIN
      value: 10.173.33.115,10.173.33.128,10.173.33.147
    image: hub.c.163.com/comb/cw-zc-admin:v1.12.1
    imagePullPolicy: IfNotPresent
    name: zc-admin-a
  resources:
    limits:
      cpu: "1"
      memory: "2109128704"
    requests:
      cpu: "1"
      memory: "2109128704"
  securityContext:
    capabilities:
      add:
      - NET_ADMIN
  volumeMounts:
  - mountPath: /app-logs/
    name: log-volume-1-1
  dnsPolicy: ClusterFirst
  nodeSelector:
    stateful: "false"
  restartPolicy: Always
  volumes:
  - hostPath:
```

```
path: /container/logs/cw-app-online/zc-admin-a/app-logs/  
name: log-volume-1-1
```

5.5.5 微服务测试

本节我们讨论针对微服务的测试。首先介绍微服务架构的特点及测试挑战，然后按照测试类型来分析这些技术是否都适用于微服务架构，以及需要做出哪些适配才能更好地为微服务架构服务。

1. 微服务架构的特点及测试挑战

微服务架构就是组件化、模块化，每个组件或模块称为产品中的一个服务。不同的服务很有可能由不同的开发人员甚至不同的团队来开发并维护。针对一个产品内部不同的服务间协作的测试尤为重要，关注点在于服务内部的逻辑及服务之间的接口协议。尤其是在微服务架构中服务数量逐渐增多的场景，如何既快速验证独立服务的内部逻辑，又保证服务之间能够按照约定的协议通信并正常提供外部应用，是一个值得思考的问题。

微服务架构是针对用户功能的拆分粒度比较细。界面展示的一个功能往往涉及多个服务，从场景层面去测试是一个非常好的选择，但是对于微服务架构产品来说，场景层面测试过程中的问题如何快速定位到不同的服务中并且快速响应也是一个痛点和难点。一般来说有效的组织架构和分工可以在一定程度上解决和避免定位效率低的问题。同时要求做系统测试或者端到端测试的测试人员本身需要对微服务架构中不同服务的职责和定位有全局性的认识。

2. 测试类型

以一个 Web 应用且后端比较复杂的系统为测试对象来分析，测试类型大概有以下几种：单元测试、接口测试、集成测试、系统测试、专项测试、UI 前端测试和探索性测试等。不同维度有不同的测试类型的划分，但是基本不外乎于以上几种。

每种类型都会有扩展，例如专项测试包括异常、稳定性、性能等专项；UI 前端测试包括兼容性测试、基于视觉的测试、基于组件的测试等；探索性测试内容更为丰富，惠特克所著的《探索式软件测试》专门描述了这类测试方法。

专项测试、UI 前端测试、探索性测试在微服务或者非微服务项目的实施上差异不大，

在有限的时间和人力下，我们在此重点分析单元测试、接口测试、集成测试、系统测试在微服务架构下的适配及实施侧重点。（注：每个名词的解析可以访问 TheTestingMap.org 来获取相关内容。）

单元测试、接口测试、集成测试、系统测试的层次关系可以用图 5-32 的测试金字塔来表示。

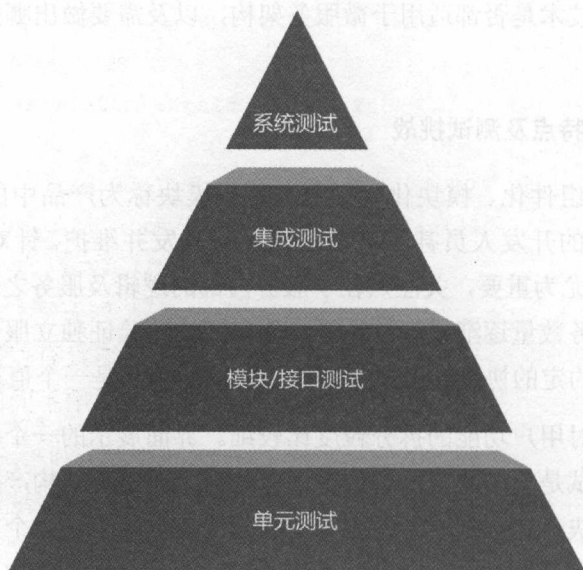


图 5-32 测试金字塔

单元测试

单元测试，是指对软件系统中最小的集合例如方法或者函数级别进行测试和验证。单元测试是最低级别的测试活动，这个阶段重点在于验证一个函数的输入输出是否符合设计，而不是验证功能或特性，也不是测试服务提供的接口。TDD（Test-Driven Design，测试驱动开发）的理念就是先写单元测试的用例，然后再实现逻辑业务代码。不过在互联网真实产品中，这种方式应用得很少。一般来说，越底层的产品会越重视单元测试，因为在单元测试发现的 bug 修复成本是最低的，而底层产品相对来说比上层产品的质量要求或者系统稳定性要求更高，因此单元测试是底层产品非常必要的一件事情。底层产品和上层产品的案例可以参考一个分布式文件系统和一个移动端 APP（例如交换闲置物品的一个 APP），单元测试的重视程度肯定是前者更高一些。

在单元测试中一般会将测试代码直接集成到业务代码库的方式来维护和运行，比如使用 Java 语言并且使用 Maven 工程的业务代码都会有现成的 test 的结构提供出来让开发者直接添加，并且可以使用一些现成的代码或者用例管理框架（JUnit 或者 TestNG）来组织自己的测试代码。一般单元测试运行的时候不会真正启动服务，而是采用直接运行业务代码的函数方式来开展测试，假如业务代码依赖其他模块或者服务，则可以使用 Mock 的方式做，尽量做到单元测试执行的时候不依赖其他模块或函数。Mock 方式很多，Java 中可以用 Jmock 来完成。

单元测试的目的是在最早期排查业务代码中比较初级的 bug，并且维持系统一个健康的生态圈，在后续代码不断修改和提交后可以保证最基础范围的质量（例如后面的开发不小心修改了前人开发的逻辑导致单元测试失败，就需要排查是否是真正的设计需要还是不小心引入的 bug）。相比于有单元测试的项目来说，无单元测试的项目更像是一个裸奔在大街上的人，毫无安全感。

微服务的单元测试编写方法及技术跟传统的单元测试基本没有差别。唯一的区别就是微服务的单元测试分布在不同的工程中，运行的框架也可以采用不同的方式。例如多个微服务架构可能有 Java 语言、Python 语言或 Go 语言编写的，他们的单元测试也需要用和业务代码一致的方式和框架来编写。执行的时候会跟工程代码编译的时候触发，在服务自身的范围方面，单元测试能起到相应的质量保障作用。

接口测试

接口测试，是对系统内一个模块或者一个独立服务对外提供的接口进行检查和验证。一般来说接口测试在单元测试之后和集成测试之前运行。如果是分层系统，则底层模块或服务先做接口测试，然后再由调用模块进行接口测试，最终再进行集成及系统测试，保证完整的系统可用。

一个模块会依赖于其他的模块与服务，为了隔离或者排除其他依赖的模块或服务的影响，常用的方式是将依赖的模块或服务打桩。从而给被测模块或服务返回预期的响应。或者可以采用 Mock 的方式。打桩和 Mock 的区别可以理解为打桩一般会硬编码一些输入和输出，比较适用于简单场景下应用。而 Mock 除了可以模拟返回预期的响应外还可以深入模拟模块之间的交互方式，例如调用次数与返回响应之间的关系，在某种情况下模拟抛出异常等复杂场景。

接口测试如果针对于系统最外层暴露的接口的话，某种意义上也可以看作集成测试和系统测试的一部分。接口测试的执行可以借助的工具和方法有很多，例如 HTTP 接口可以用 Postman 工具来构造并且传送，也可以直接在 Java 语言中使用 HttpClient 类来实现接口的各种 POST、PUT、GET、DELETE 等方法。

接口测试的最终目的就是验证模块或者服务提供出去的接口内部的逻辑符合设计的预期，一般设计接口测试的用例都会考虑 MBT (Model Based Test) 测试设计法，即将接口的输入值及返回响应码分别通过等价类方法和组合法来拼装成不同的用例，并且确保用例走过的代码路径可以覆盖大部分的应用场景。通过将用例结果与设计或者用户预期对比来判断该接口是否能够通过测试。

微服务架构下的接口测试需要多利用打桩和 Mock 的方式降低不同服务的相互依赖。并且微服务架构下接口测试语言类型或者接口协议同样可以有多种，需要不同的技术手段来实现接口的调用和数据模拟。

集成测试

集成测试是在单元测试、接口或模块测试的基础上，将所有模块或者子服务拼装成一个子系统或者系统，进行集成后的测试。如上文所说，如果接口测试针对系统的最外层暴露的接口进行，也可以看作集成测试的一部分。集成测试执行的前提已经确保各个子模块或者服务本身没有问题，重点验证不同模块或者不同服务之间的交互存在的潜在问题。可以用一些通过数据流走向覆盖法测试来进行测试设计和分析，从而输出针对性的集成测试的用例。

在微服务架构系统中，由于服务数量比较多，集成测试会有一部分时间花在系统内部多个服务的更新构建上面，如何高效有顺序地构建并部署，是一个微服务架构系统的维护人员及测试人员日常特别需要关注的难题。本书的 4.6 节“持续集成”会提到有效解决快速高效更新和部署的方案。当部署完毕后，借助将接口测试和集成测试的自动化代码集成到持续集成方案的 Pipeline 上，自动执行测试，从而降低由于代码更新导致不停集成测试和回归的测试成本。

系统测试

系统测试，又叫端到端测试，指的是将所有系统中的模块或者服务都整合起来，整体

呈现给用户，对整个系统在用户场景下从功能和非功能角度出发进行的测试活动。

在系统测试中，建议所有准备的用例数据来源都跟实际用户尽量真实一致。系统测试的验收采用的测试方法以黑盒测试为主，比如 IBO (Input-Box-Ouput) 测试设计法，重点关注用户的输出和最终产生的结果是否符合预期。另外稳定性和性能等专项测试也可以在这个阶段完成。最终输出的质量评价体现了系统或者产品的总体质量。

系统测试处于测试金字塔的顶端，也就是说测试的代价和花费成本最高。因此在微服务架构下如何高效地进行系统测试是一个很大的难题和挑战。

3. 解决微服务结构测试的技术和方法

在一个单体架构服务里，一般会有 Web 服务层（用户界面）、业务逻辑层（处理真实的业务）和数据层。业务逻辑的复杂化、微服务化改造之后，可以根据不同的业务拆分为不同的服务，例如登录服务、产品服务、日志服务、报表服务等。拆分后的模块可以共用一个 Web 层，但后端都分属于不同的微服务，数据也可以在不同的数据库或者不同的存储媒介中。例如日志服务实时要求比较强，可以采用 Redis 存储。登录服务安全要求比较高，一致性要求比较高，可以用传统的关系数据库存储。而产品服务会与其他几个服务之间有一些调用关系，如果一个产品会有对应用户不同的权限，就需要调用登录服务中的权限管理来检查，产品里面的日志或者报表会调用日志服务和报表服务来进行存储和更新。这样就构成了一个典型的微服务架构。

开发人员可以根据用户的需求对不同的微服务模块内部进行代码开发和提测。测试人员拿到需求以后首先需要分析这个需求涉及的服务范围和影响程度，对用户直观展现的情况做具体的测试策略和测试分析。最终的需求验收如果仅仅从用户角度去关注功能点和界面输出，是非常容易的，然而对于一个复杂的、微服务化的产品来说，用户级别的需求验收远远不够，也许页面输出看上去很美，但系统内潜藏的稳定性隐患、异常分支带来的各种异常无法关注到，等到引发更为严重的数据不一致、用户数据丢失等严重 bug 则为时已晚。微服务级别的验收对应于原来的模块测试来说，既需要继承模块测试中的一些方法和手段：例如 MBT（基于模型的方法 Model Based Test）、基于数据流的方法、基于覆盖模式的方法等，又需要引入一些微服务特有的针对性测试，例如想关注服务本身逻辑，而不想过多关注其依赖的其他微服务接口，可以用 Mock 方法来完成。

Mock 方法应用于微服务架构测试

拿上文中提到的典型微服务架构产品作为被测对象，产品服务中有一个将某公开产品收藏为自己常用产品的功能，它需要先调用登录服务的权限管理来获得产品所属的用户组的权限，并且根据不同的权限来判定是否可以收藏，如果使用完整的微服务架构的方式，需要把全部的服务都部署好，然后在登录服务的数据存储中构造不同权限的用户，并且结合产品服务的逻辑去覆盖相应的分支。如果使用 Mock 方法，则不需要依赖完整的登录模块的部署和数据构造。只需要定义 Mock 模块的输入和输出返回，就可以构造想要的不同权限的返回值。

业界使用 Mock 的工具和源码也很多，例如在 GitHub 上就可以找到 136 个关于 MockServer 的实现。比如实现一个针对自己产品的 MockServer，如果需求比较简单，则可以用现成的 Mock 工具初步模拟。常用的 Mock 框架有 EasyMock、GMock 等，也有提供独立 Jar 包直接运行的方式，例如 Moco、MockServer-Netty 等，原理上也比较类似，都是通过模拟一个服务的输入和输出，达到让被测服务覆盖不同分支的目的。

MockServer 使用原理

Mock 之前和之后的真实用户操作流程分别如图 5-33 和图 5-34 所示。

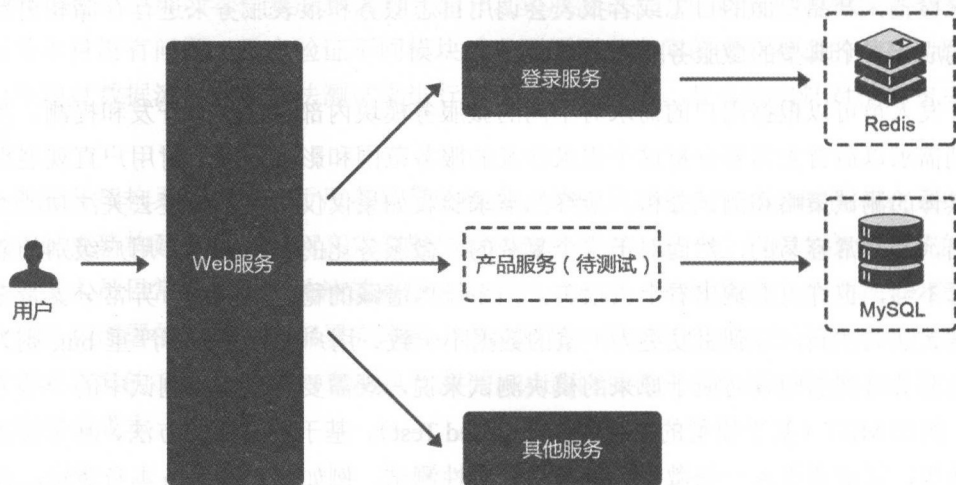


图 5-33 Mock 之前的用户操作流程

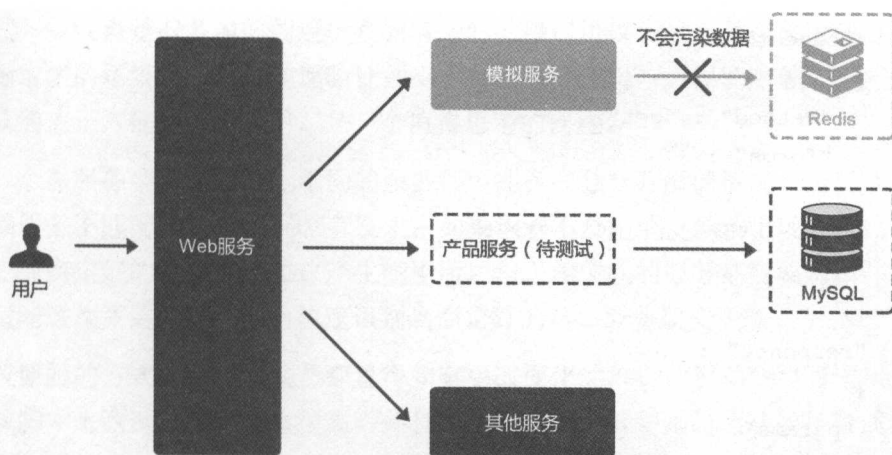


图 5-34 Mock 之后的用户操作流程

假设登录服务的地址为 10.10.1.111，它提供的其中 1 个接口为 GET 10.10.1.111:8181/getOpenLevel?productId=123，预期返回值 json 格式如下。

```
{
  "params":
  {
    "openlevel": "true"
  },
  "code": 200
}
```

返回值有以下几种。

```
code 200 openlevel 为 true;
code 200 openlevel 为 false;
code 400 (productId 查不到记录);
code 500 (登录服务内部错误);
```

注：接口和返回参数都经过简化，以能说明流程为主。如果不使用 Mock 就想测试全面，则需要构造数据库中不同的产品及用户的数据，并且很难模拟出 500 错误。

使用 MockServer 的步骤如下（以 Moco 为例）。

1. 首先定义好预期的输入输出 json 模板，例如 test.json。

```
{
```



```
"request" :
{
  "method" : "get",
  "forms" :
  {
    "productId" : "123"
  }
},
"response" :
{
  "params":
  {
    "openlevel": "true"
  },
  "code": 200
}
```

2. 启动 Mock 程序, 这里是一个 Jar 包(注: 下载地址为 <https://repo1.maven.org/maven2/com/github/dreamhead/moco-runner/0.11.0/moco-runner-0.11.0-standalone.jar>), 将 json 配置导入。

```
java -jar moco-runner-<version>-standalone.jar start -p 8181 -c test.json
```

并且查看 Mock 程序所在的服务器地址: 10.10.1.112。

1. 被测系统调用的登录服务的地址配置改成 10.10.1.112:8181, 运行被测系统, 就可以拿到想要返回的响应。

2. test.json 支持在线修改和导入, 不需要重启 Mock 程序, 增加了灵活性。

通过上述步骤可以较完整地覆盖返回值的 4 种场景, 而不需要污染任何登录服务相关的数据, 也不需要做系统故障模拟等成本较高的异常模拟测试。当然这里的举例比较简单, 逻辑复杂的场景需要更多用例分析来覆盖, 并且可以采用自定义灵活的 MockServer 的方式来模拟输入输出。大体思路万变不离其宗, 基本可以满足大部分微服务架构场景下的测试需求, 在微服务架构中自动识别接口协议变更的测试。

上文中讨论到微服务架构下测试面临的挑战之一是针对一个产品内部不同的服务间协

作的测试，关注点是服务内部的逻辑及服务之间的接口协议。尤其是在微服务架构中服务数量逐渐增多的场景，如何既快速验证独立服务的内部逻辑，又保证服务之间能够按照约定的协议通信正常提供外部应用，是一个值得思考的问题。

在一个微服务架构产品内，不同的服务都可能存在独立升级的情况，接口或协议变更导致依赖服务不匹配，这种情况时有发生，如果沟通不及时就会导致出现系统性 bug。因此一旦之前商定好的接口或者协议产生变更后，与之相关联的服务都需要感知并且做出调整，以适配这次变更。那么如何快速识别到商定好的接口或协议变更呢？

比较原始的方式是从开发人员那里得知有更改变化的接口，手动变更测试用例及自动化代码或脚本来测试。这种成本很高，并且有遗漏风险，特别是跨团队的接口或者协议变更往往不一定能及时知会到测试人员。这个时候就需要一套自动化识别变更的机制或工具，接口自动化管理工作流程如下。

1. 分析系统中现有的接口并且入库：拿一个 HTTP 接口举例，需要将接口的 Header 和发送数据、响应 Header 和响应数据的格式都分类并且记录到数据库中。
2. 接口测试：对接口的参数进行验证和测试，并且将自动化脚本入库。
3. 脚本管理：脚本分类存储，可以分策略分批或者单个拉起执行。
4. 更新接口：人工或者自动更新接口。
5. 监控及结果输出：结果推送、报警、发送邮件等内容。

首先我们需要维护微服务架构产品中原有的接口，并且保证大部分接口都上传了相应的测试脚本，策略是定期执行。当不同的服务提交了代码可能会影响服务往外提供的接口或者协议时，自动化识别变更的工具就会扫描最新的接口，并与目前数据库中存在的老接口进行自动化对比，具体流程如下。

1. 开发人员提交代码。
2. 自动识别代码提交开始全量扫描获取最新接口。
3. 提取数据库中的接口模板，对比 API 的几个要素（URL、请求参数、响应等）。
4. 对比 URL 及参数是否相同，如果相同则不修改库信息，如果不同或者库中不存在则进入自动识别逻辑。
5. 自动识别逻辑：如果 URL 没有，则是一个新接口，直接入库，报警并且测试人员

手动添加接口参数及自动化脚本。

6. 自动识别逻辑：如果 URL 有，但是响应码不正确，则接口异常，不处理。

7. 自动识别逻辑：如果 URL 有，响应码与库中一致，对比响应体是否一致，一致则不处理，不一致直接入库，并且报警告知测试人员修改自动化脚本。

测试人员收到接口或协议变更提醒后，上去查看入库的新接口或接口修改的参数变更，对自动化脚本进行添加或修正，从而保证快速识别接口变更及保证质量的职责。

4. 微服务测试自动化集成思路和实践

不管是单元测试、接口测试、集成测试还是系统测试，大量重复的手工操作都会让测试工作效率低下。自动化手段是首当其冲要考虑的事情。对于单体架构的系统，测试自动化可以以持续集成中部署系统后面的一个环节集成进去。微服务架构下的自动化集成相对复杂一点，需要考虑多个服务的自动化及集成后统一对外的自动化的顺序关系，并且需要考虑执行效率，因为服务拆分会导致服务间提供的接口非常多，因此把所有的自动化测试都放在一个环节或者一个 Job 内完成不是最优的选择。单体架构系统的自动化集成工作流（应用容器技术）如图 5-35 所示。



图 5-35 单体架构系统的自动化集成工作流（应用容器技术）

预编译：代码提交触发代码构建并且得到二进制文件或者运行包。

编译：将二进制文件复制到容器中，并且保存到镜像，提交给镜像仓库。

集成测试：拉取镜像，启动容器，通过配置控制容器运行在线下环境，启动容器内服务，对服务进行测试（手工或者自动化）。

预发布：代码测试完毕和 bug 修复后，再次提交镜像。

发布：将可以发布的镜像更新部署到线上环境，供外部使用。

微服务架构系统的自动化集成工作流（应用容器技术）如图 5-36 所示。

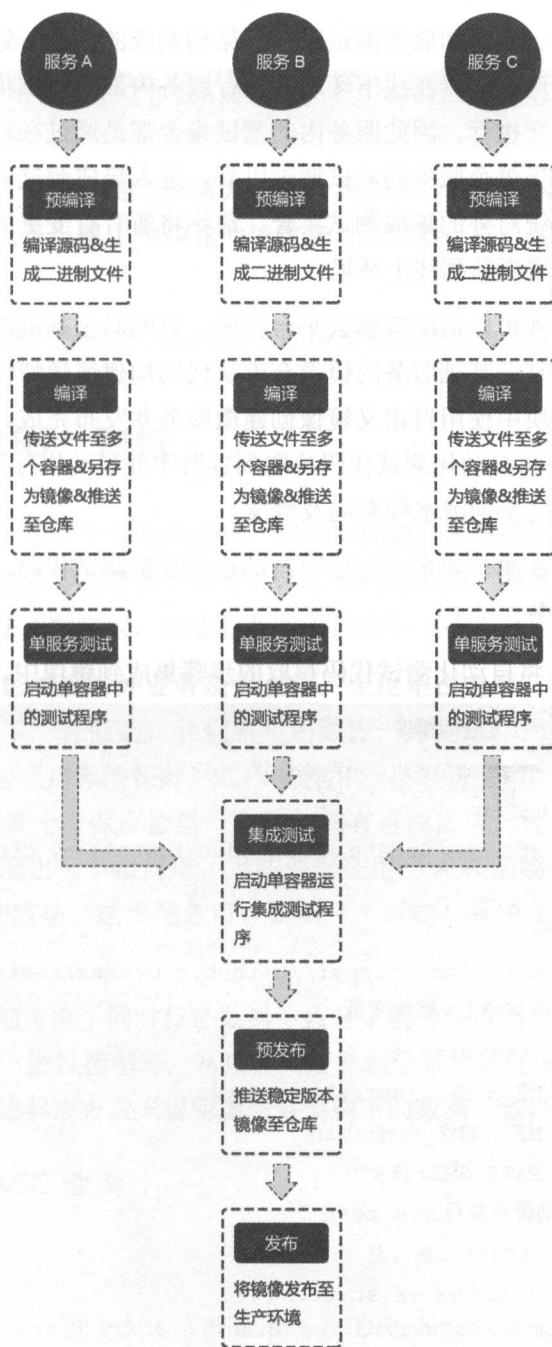


图 5-36 微服务架构系统的自动化集成工作流（应用容器技术）

每个服务的预编译、编译和服务测试跟单体架构系统的流程一致，都是先通过代码提交触发构建镜像，然后运行容器在线下环境并进行服务内部的测试集合。这时需要注意不同的服务可能会并发提交执行，因此服务内的测试集合都是通过在独立的容器中拉起运行并且收集运行日志的。当单个服务的测试通过以后，进入集成测试的环节。这时一个或者多个容器会运行整体系统对外的集成测试集合。最终将所有有变更的服务所在的镜像都提交一遍到镜像仓库，部署更新到线上环境。

上述 workflows 中的步骤可以借助容器云平台完成，以网易云基础服务平台举例，预编译和编译步骤可以运用网易云基础服务的镜像仓库从代码构建镜像特性完成。发布步骤可以在平台内的服务管理模块中使用自定义镜像创建微服务并发布完成。集成测试则可以使用编写 Dockerfile 的方式，将自动化测试代码拉取到容器中并自动执行。网易云基础服务的持续集成特性也可以完成完整的流水线布局及触发。

集成测试步骤如何在单个或多个容器中自动运行批量测试代码，并且收集日志？以下的实践可以完成这项任务。

1. 编写 DockerFile 将自动化测试代码拉取的步骤集成到镜像中。

```
#拉取 JDK1.7 官方基础镜像
FROM          hub.c.163.com/public/jdk:1.7.0_03
#安装 git 和 maven
RUN          apt-get update && apt-get install -y git && apt-get install
-y maven
#拉取测试代码
RUN          git clone https://github.com/xxxx/test-code.git
#设置工作目录和 MAVEN 环境变量
WORKDIR      /test-code
ENV          M2_HOME /usr/share/maven
ENV          M2 $M2_HOME/bin
ENV          PATH $M2:$PATH
#创建容器启动脚本执行 mvn test
ADD          start.sh ./
RUN          chmod +x start.sh
#设置容器入口
ENTRYPOINT   sh start.sh && /usr/sbin/sshd -D
```


其中 start.sh 内容如下。

```
mvn clean test "-Dmaven.test.failure.ignore=true"
"-DsuiteXmlFile=/test-code/xml/testng.xml" 1>&2
```

2. 拉起单个或多个容器，通过配置方式传入需要运行的测试执行集合配置。使用服务管理中的自定义镜像选择步骤 1 中构建的镜像来启动容器，并且配置好传入的参数及收集的日志路径。

3. 收集日志并反馈执行结果。使用日志服务将测试结果收集起来并且分析是否通过。

5.6 分布式数据一致性

微服务架构本质上是一个分布式系统，服务间的通信本质是以数据复制的方式进行交换的，由于网络或者服务存在诸多不确定性，比如网络故障、网络延时、服务异常等，使得数据复制的过程无法正常完成，造成服务间的数据不一致。

数据不一致直接影响到对外业务是否正常，无论是在电商、游戏、金融等领域。例如，用户从网上购买了一件商品，我们希望的数据一致是这样的，当用户支付成功后，商品库存减 1。如果这一点无法保障，就意味着用户花了钱没买到东西。或者两个用户同时购买一件商品，如果无法保证数据一致性，就有可能出现一件商品被同时卖给了两个人。如何避免这种情况出现？很简单，系统要保证用户 A 在购物过程中，用户 B 必须等着，直到用户 A 购物成功。这个场景可以说明一个问题，系统如果想要保证一致性，可能影响性能。

如何在保证性能的情况下同时保证数据一致是个值得探讨的话题，本节首先介绍 CAP 原理和 BASE 理论、一致性模型等，从理论角度介绍分布式系统设计的关键要素，再通过几种工程中用得较多的解决方案来说明微服务架构下的数据一致性实践。

5.6.1 CAP 和 BASE 理论

CAP 理论

2000 年，Eric Brewer 在 PODC（Principles of Distributed Computing）会议上提出了著名的 CAP 理论，如图 5-37 所示。2 年后，Seth Gilbert 和 Nancy Lynch 从数学上证明了这一

理论。CAP 理论告诉我们，在分布式系统中，一致性（Consistency）、可用性（Availability）和分区容忍性（Partition Tolerance）3 个要素最多只能同时满足两个，不可兼得。

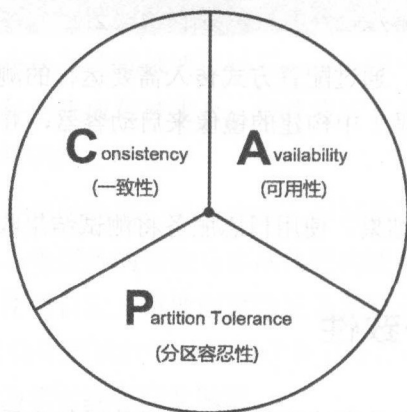


图 5-37 分布式架构设计的 CAP 原理

- 一致性：分布式环境下多个节点的数据是否强一致。
- 可用性：分布式服务能一直保证可用状态。当用户发出一个请求后，服务能在有限时间内返回结果。
- 分区容忍性：特指对网络分区的容忍性。

在分布式系统中，分区容忍性不可或缺，否则就不是一个分布式系统了。下面通过一个例子介绍分布式数据一致性的两种方案。假设我们的服务分别部署在两个数据中心，每个数据中心的实例都有一个数据库，同时，两个数据库会互相进行数据同步，即应用实例对数据库的读写操作都落在本地数据库，然后由数据库的同步机制对两个节点进行数据同步，如图 5-38 所示。

1. 牺牲一致性

如果我们要优先保证系统可用性。DC1 和 DC2 间出现网络故障，DC2 上的应用实例将看不到 DC1 上的数据变更，DC1 上的应用实例也看不到 DC2 上的数据变更。就是说，我们的系统仍然可用，两个 DC 的应用实例在网络分区后仍然能够响应服务请求，但是失去了数据一致性。这样被舍弃了一致性的系统被称为一个 AP 系统。

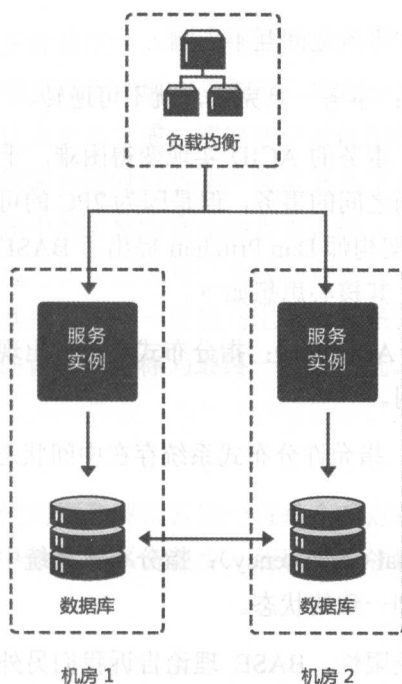


图 5-38 分布式系统的数据同步流程

2. 牺牲可用性

如果我们要优先保证系统一致性。每个 DC 的数据库都需要知道本地数据副本和其他数据库节点副本的数据要完全一致。在网络分区的情况下，数据库节点之间无法通信，也就无法同步数据。要保证系统的数据一致性，我们唯一的选择只能是应用拒绝响应请求。这样被舍弃了可用性的系统被称为一个 CP 系统。

CAP 理论在 NoSQL 中应用很广，比如 Cassandra、Dynamo 等，默认优先选择 AP，弱化 C，对一致性要求低一些。另外一些比如 HBase、MongoDB 等，默认优先选择 CP，弱化 A。

BASE 理论

传统关系型数据库通过事务来保障数据的 ACID 特性。

- 原子性 (Atomicity)：事务中所有操作要么全部完成，要么全部不完成。
- 一致性 (Consistency)：事务开始或结束时，数据库处于一致状态。

- 隔离性 (Isolation): 事务之间互不影响。
- 持久性 (Durability): 事务一旦完成, 就不可逆转。

在数据库分区出现之后, 事务的 ACID 实现变得困难, 于是厂商引入 2PC (两阶段提交) 协议来保障数据库多实例之间的事务, 但是因为 2PC 的可扩展性很差, 在分布式架构下应用代价较大, 于是 Ebay 架构师 Dan Pritchett 提出了 BASE 理论, 用于解决大规模分布式系统下的数据一致性问题。其核心思想如下。

- 基本可用 (Basically Available): 指分布式系统在出现故障时, 允许损失部分的可用性来保证核心可用。
- 软状态 (Soft State): 指允许分布式系统存在中间状态, 该中间状态不会影响到系统的整体可用性。
- 最终一致性 (Eventual Consistency): 指分布式系统中的所有副本数据经过一定时间后, 最终能够达到一致的状态。

分布式事务的实现存在局限性, BASE 理论告诉我们另外一种思路: 可以通过放弃系统在每个时刻的强一致性来换取系统的可扩展性。

5.6.2 一致性模型

数据的一致性模型可以分成以下 3 类。

1. 强一致性: 数据更新成功后, 任意时刻所有副本中的数据都是一致的, 一般采用同步的方式实现。
2. 弱一致性: 数据更新成功后, 系统不承诺立即可以读到最新写入的值, 也不承诺具体多久之后可以读到。
3. 最终一致性: 弱一致性的一种形式, 数据更新成功后, 系统不承诺立即可以返回最新写入的值, 但是保证最终会返回上一次更新操作的值。

分布式系统数据的强一致性、弱一致性和最终一致性可以通过 Quorum NRW 算法分析, Quorum NRW 是一种乐观的保证分布式系统一致性的算法, 通过对读写副本数的不同设值来获得性能、一致性和可用性之间的平衡, 感兴趣的读者可以通过 [https://en.wikipedia.org/wiki/Quorum_\(distributed_computing\)](https://en.wikipedia.org/wiki/Quorum_(distributed_computing)) 了解它的具体算法思想。

举个实际的例子，比如支付系统中，买家执行完支付 100 元的操作之后，就认为完成了一次购买行为，这 100 元要从买家的账户里扣除，实时加入卖家的账户里，或者说，只有卖家账户收到钱了，才能认为交易完成，买家账户才能扣钱，这就是数据强一致系统。另外比如比价系统，比价系统后台定时任务定时抓取其他平台的商品价格，定时任务可能抓取失败，这就导致比价系统中的价格与商品的当前价格一直不一致，只要比价系统能接受这种不一致，我们就可以认为比价系统是一个弱一致性的系统。再比如评价系统，买家对一件商品评价，卖家或者其他买家不一定能马上看到这条评论，只有过了下个数据同步周期才能看到，我们把这里的评价系统称为最终一致性系统。

5.6.3 典型的解决方案

分布式系统的数据一致性问题 and 解决方案一直是个难点和技术热点。上节提到的数据一致性模型分类，能接受弱一致性的系统很少，所以工程领域主要讨论的还是强一致性和最终一致性的解决方案。

1. 强一致性解决方案

解决强一致性问题可以用分布式事务解决方案，比如两阶段提交方案。

两阶段提交（2PC, Two-phase Commit）方案

通过第 4 章的介绍，我们了解到 2PC 实际上是一种协议。在一个分布式系统里，由于网络分区的存在，每个节点（参与者）只能确切地知道自己发出的操作结果成功或者失败，但无法确切知道其他参与者节点操作结果是否成功。当一个事务跨越了多个参与者节点时，为了保证事务的 ACID 特性，需要引入一个协调者节点来统一掌管所有参与者节点的操作结果，决定这些参与者节点是否要提交操作结果状态，比如是否把更新完毕的数据持久化到磁盘。因此，2PC 的算法思路可以简单概括为：所有参与者节点将操作是否成功状态通知给协调者节点，再由协调者节点根据所有参与者节点的反馈结果来决定各参与者节点是否要提交最后的结果。

接下来对 2PC 算法的两个阶段进行更详细的说明。

第一阶段（提交请求阶段、Prepare 阶段、投票阶段）

1. 协调者节点向所有参与者节点发送一个请求执行提交操作的消息，并且一直等待各

参与者节点的响应消息。

2. 所有参与者节点开始执行从协调者节点发起请求到当前为止的所有事务操作，并且分别将 Undo 和 Redo 信息写入日志。

3. 各参与者节点分别响应协调者节点发起的请求消息。如果参与者节点的事务操作执行成功了，它就返回一个“同意”的响应消息；如果参与者节点的事务操作执行失败了，它就返回一个“中止”的响应消息。

第二阶段（提交执行阶段、Commit 阶段、完成阶段）

如果协调者节点从所有参与者节点收到的第一阶段响应消息都为“同意”，就会出现以下情况。

1. 协调者节点向所有参与者节点发送一个“正式提交”的消息。

2. 各参与者节点完成操作，并释放在事务期间占用的锁和资源。

3. 各参与者节点分别向协调者节点发送“完成”的确认消息。

4. 协调者节点收到所有参与者节点响应的“完成”消息后，完成事务。

如果有任意一个参与者节点在第一阶段返回的响应消息为“中止”，或者协调者节点在等待第一阶段参与者节点的响应消息时出现了超时，就会出现以下情况。

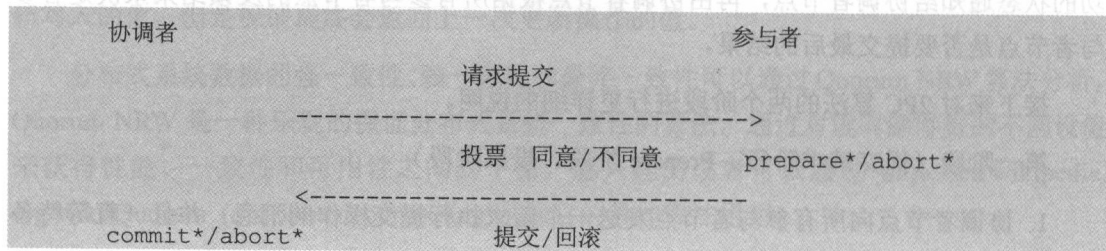
1. 协调者节点向所有参与者节点发出“回滚”的消息。

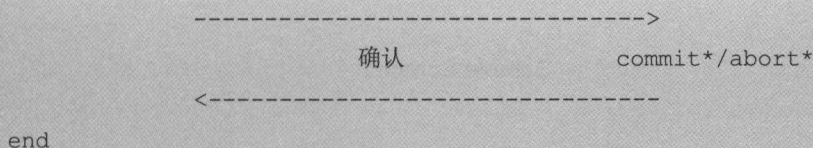
2. 各参与者节点利用之前写进日志的 Undo 信息执行回滚操作，并释放在事务期间占用的锁和资源。

3. 各参与者节点分别向协调者节点发送“回滚完成”的确认消息。

4. 协调者节点收到所有参与者节点响应的“回滚完成”消息后，取消事务。

两阶段协议中协调者和参与者之间的通信流程如下。





两阶段提交方案的问题分别如下。

1. 同步阻塞。每个参与者节点都有锁资源操作，资源锁住期间，其他访问该资源的节点也会被阻塞住，导致性能问题。
2. 数据不一致。在第二阶段里，一旦协调者和某些参与者节点之间出现网络问题，会造成部分参与者节点提交了事务，而另一部分没有提交，从而造成参与者节点之间的数据不一致。
3. 单点问题。协调者在协议中处于核心位置，是个单点，如果协调者挂掉，会导致参与者节点一直处于某个阶段中无法恢复。

为了解决两阶段提交方案的这些缺陷，学者们又提出了三阶段提交方案，当然三阶段提交也存在一些缺陷，要彻底从协议层面避免数据不一致，可以采用 Paxos 或者 Raft 算法，相关算法介绍可以参考以下资料：[https://en.wikipedia.org/wiki/Paxos_\(computer_science\)](https://en.wikipedia.org/wiki/Paxos_(computer_science))和<https://raft.github.io/>。

2. 最终一致性解决方案

(1) eBay 事件队列方案

eBay 的架构师 Dan Pritchett，曾在一篇解释 BASE 原理的论文《Base: An Acid Alternative》中提到一个 eBay 分布式系统一致性问题的解决方案。它的核心思想是将需要分布式处理的任务通过消息或者日志的方式来异步执行，消息或日志可以存到本地文件、数据库或消息队列，再通过业务规则进行失败重试，它要求各服务的接口是幂等的。这个方案就是保证最终一致性的解决方案，业界后来根据自身的业务特征又衍生出多种变种方案。

eBay 方案中描述的场景为，有用户表 user 和交易表 transaction，用户表存储用户信息、总销售额和总购买额，交易表存储每一笔交易的流水号、买家信息、卖家信息和交易金额，如图 5-39 所示。如果产生了一笔交易，需要在交易表增加记录，同时还要修改用户表的金额。由于这两个表分别属于用户服务和交易服务两个独立的服务，所以涉及分布式事务一

致性的问题。

Sample Schema

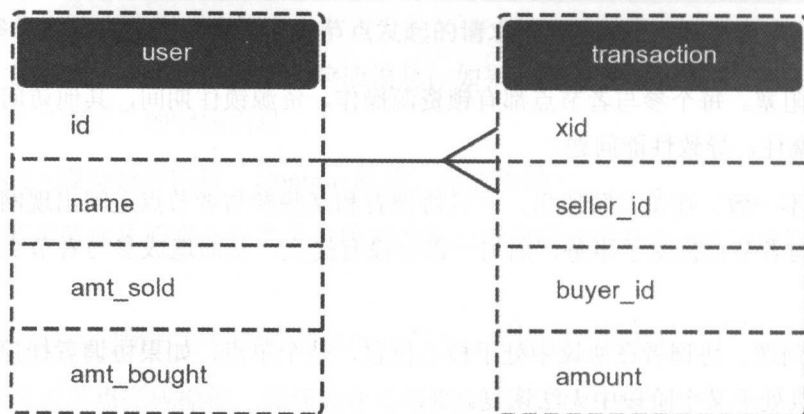


图 5-39 eBay 事件队列方案场景

论文中提出的解决方法是将更新交易表记录和用户表更新消息放在一个本地事务来完成，为了避免重复消费用户表更新消息带来的问题，增加一个操作记录表 `updates_applied` 来记录已经完成的交易相关的信息，如图 5-40 所示。

Update Table

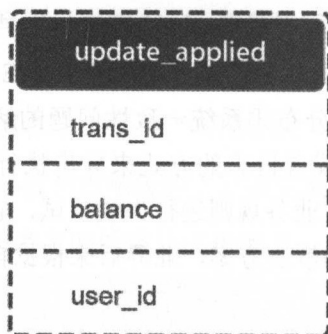


图 5-40 增加一个操作记录表 `updates_applied` 来记录已经完成的交易相关的信息

算法关键部分的伪代码如下。

```

Begin transaction
    Insert into transaction(id, seller_id, buyer_id, amount);
    Queue message "update user("seller", seller_id, amount)";
    Queue message "update user("buyer", buyer_id, amount)";
End transaction
For each message in queue
    Peek message
    Begin transaction
        Select count(*) as processed where trans_id = message.trans_id
        and balance=message.balance and user_id = message.user_id;
        If processed == 0
            If message.balance == "seller"
                Update user set amt_sold + message.amount where id =
message.id;
            Else
                Update user set amt_bought + message.amount where id =
message.id;
            End if
            Insert into updates_applied(message.trans_id, message.balance,
message.user_id);
        End if
    End transaction
    If transaction successful
        Remove message from queue
    End if
End for

```

第一阶段，通过交易服务的本地事务保障交易记录入库，以及用户表更新（买家、卖家）消息发送到消息队列。第二阶段，用户服务从消息队列读取用户表更新消息，启动本地事务，通过操作记录表 `updates_applied` 来检测相关交易是否已经完成，未完成则会修改 `user` 表，然后增加一条操作记录到 `updates_applied`，事务执行成功之后再删除队列消息。该方案通过两阶段多个本地事务的组合，实现了分布式系统的最终一致性。

可以看到这个方案的核心在于第二阶段的重试和幂等执行。失败后重试，这是一种补偿机制，它是能保证系统最终一致的关键流程。但是重试操作有一个前提，就是重试操作本身不能引起副作用，即重试对象要支持幂等。所谓幂等性，是指对系统的一次请求和多

次请求返回结果一样，多次请求不会使得系统内部状态不一致。

(2) TCC 补偿模式

补偿模式是一种比较容易实现的数据一致性解决方案。某业务模型如图 5-41 所示。

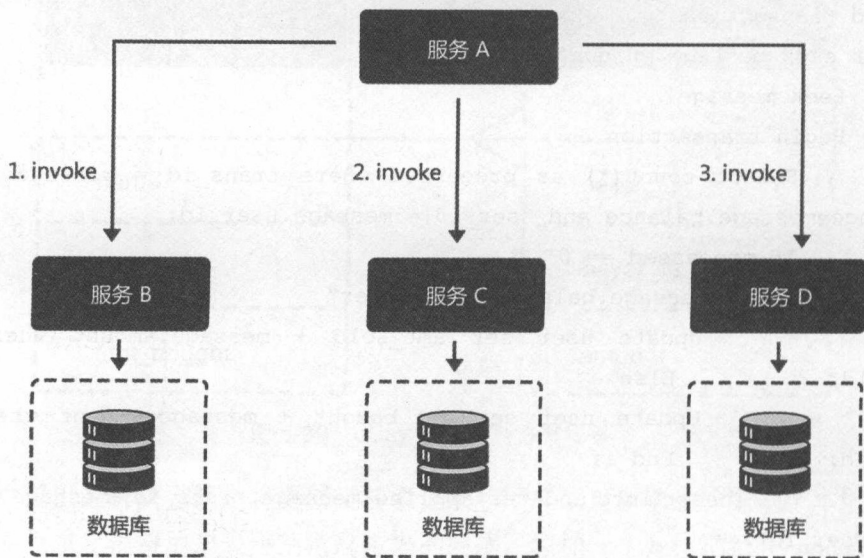


图 5-41 某业务模型

由服务 A、服务 B、服务 C、服务 D 共同组成的一个微服务架构系统。服务 A 需要依次调用服务 B、服务 C 和服务 D 共同完成一个操作。当服务 A 调用服务 D 失败时，若在保证整个系统数据的一致性，就要对服务 B 和服务 C 的 invoke 操作进行回滚，执行反向的 revert 操作，如图 5-42 所示。

回滚成功后，整个微服务系统是数据一致的。补偿模式的实现有以下几个关键要素。

1. 服务调用链必须被记录下来。服务调用链的记录可以以数据库或者日志的方式落盘，由服务调用发起方记录，也可以由一个统一的第三方协调服务记录。这个调用链信息后续会被当成业务的流水记录，作为执行回滚操作的唯一依据。

2. 每个服务提供者都需要提供一组业务逻辑相反的操作，互为补偿，同时回滚操作要保证幂等。因为由服务调用发起方或者协调服务发起的回滚操作可能会出现多次。

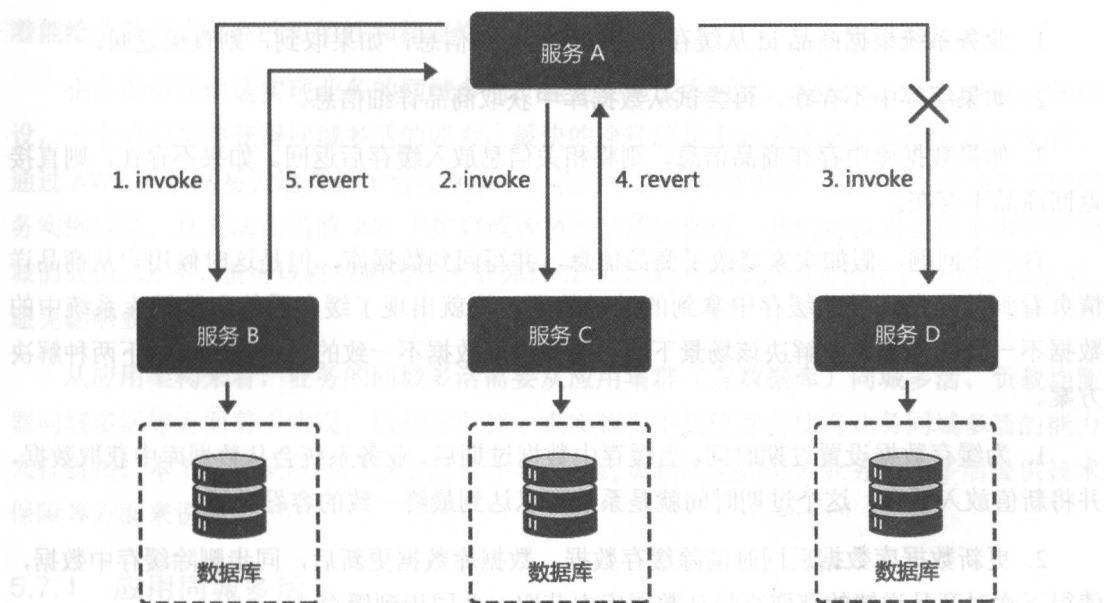


图 5-42 补偿模式

上面的讨论没有考虑服务 A 调用服务 D 失败的原因，但是恰恰这个失败原因很重要，也是实现补偿模式的一个难点。我们把失败原因分为网络故障和拒绝服务，其中网络故障可以细分为网络超时或者网络中断，拒绝服务也可能包含服务内部错误。这就要求我们必须按失败原因对服务 D 执行不同的回滚策略。比如网络中断造成的失败就不用对服务 D 执行回滚，对于服务 D 内部错误造成的调用失败，服务调用发起方或者协调服务无法获知服务 D 内部是否数据一致，因此，最合适的方式是调用服务 D 的回滚操作，由服务 D 来保证回滚操作的幂等性。

补偿模式的特点是实现简单，但是想形成一定程度的通用方案比较困难，特别是服务链的记录，因为大部分时候，业务参数或者业务逻辑千差万别。另外很多业务特征使得该服务无法提供一个安全的回滚操作。

(3) 缓存数据最终一致性

再举一个缓存相关的数据一致性问题实例，在我们的业务系统中，缓存（Redis 或者 Memcached）通常被用在数据库前面，作为数据读取的缓冲，使得 I/O 操作不至于直接落在数据库上。在这个场景里，数据库和缓存系统存在数据不一致问题。比如商品详情页获取商品信息，一般的做法如下。

1. 业务系统根据商品 id 从缓存中获取商品详细信息，如果取到，则直接返回。
2. 如果缓存中不存在，再尝试从数据库中获取商品详细信息。
3. 如果数据库中存在商品信息，则将相关信息放入缓存后返回。如果不存在，则直接返回商品不存在。

有一个问题，假如卖家修改了商品信息，并写回到数据库，但是这时候用户从商品详情页看到的信息还是从缓存中拿到的过时数据，这就出现了缓存系统和数据库系统中的数据不一致的现象。要解决该场景下缓存和数据库数据不一致的问题我们有以下两种解决方案。

1. 为缓存数据设置过期时间。当缓存中数据过期后，业务系统会从数据库中获取数据，并将新值放入缓存。这个过期时间就是系统可以达到最终一致的容忍时间。
2. 更新数据库数据后同时清除缓存数据。数据库数据更新后，同步删除缓存中数据，使得下次对商品详情的获取直接从数据库中获取，并同步到缓存。

第二种方案相对于第一种方案可以达到最终一致的时间比较短，但是代码实现稍微复杂一些。我们建议即使采用了第二种方案，也最好能设置缓存的过期时间，因为通过缓存的自动过期机制，我们可以放弃数据库和缓存操作的事务性，这样当数据库操作成功，而缓存操作失败时，我们的系统仍旧可以达到最终一致。

以上 3 个例子分别说明了在特定场景下，分布式系统实现数据最终一致的解决方案。我们在面临数据一致性问题的時候，首先要从业务需求的角度出发，确定我们对于 3 种一致性模型的接受程度，再通过具体场景来决定解决方案。

5.7 同城多活

近些年，我们经常可以看到企业因为机房电力故障或者机房网络光纤被挖断导致整个数据中心服务不可用的新闻，这对企业的产品和业务产生了严重影响。随着互联网产品对服务可用性、数据可靠性的要求进一步提高，原先的单机房集中部署模式已经不能满足要求，跨机房部署的需求越来越多。

AZ (Availability Zone, 可用域) 是指分布在同一区域或城市、物理距离较近、延时较小、设计独立 (机房、供电、网络等) 的一个或者多个数据中心。跨 AZ 的服务设计意味

着能给业务带来更高的可用性和稳定性，实现同城多活。

企业期望能快速实现业务的同城多活，但是实施成本巨大，要投入大量的基础设施建设，一个应用想要获得同城多活的能力，最快的途径就是上云或者基于采用云原生架构，通过 AWS 或者网易云这类云平台实现多 AZ 能力来实现同城多活。比如当一个 AZ 下的服务实例故障，且无法在当前 AZ 下扩容或者替换故障实例时，我们可以将实例平滑迁到同城的其他 AZ 下，也可以将故障实例上的弹性 IP 绑定到其他 AZ 的实例上，继续提供服务，避免影响业务 SLA。

从应用架构来看，业务的同城多活需要从应用集群（含数据库）同城多活、负载均衡器同城多活等方面着手实现，这些方面任一点实施得不到位都会使得业务同城多活的能力大打折扣。本节主要从应用同城多活设计、云平台负载均衡如何为业务同城多活提供技术保障等方面来说明。

5.7.1 应用同城多活

应用集群同城多活是指在同城布置多个 AZ，AZ 之间一般通过高速光纤相连，此时机房间的时延往往是毫秒级的，网络延迟一般在可接受范围内，对应用的影响可以忽略。该架构下的应用有两种部署模式。

- 应用层双活，数据库双活：两个 AZ 的应用程序同时对外提供服务，两个 AZ 的数据库也同时提供读写，每个 AZ 的应用程序读写同一个 AZ 内的数据库，两个数据库之间双向复制，通过加锁解决双向写冲突问题，该模式理论上实现了数据库多点写入，但是在实际应用过程中，尤其是在写冲突密集的业务场景中，性能下降非常大，不适用于 OLTP 系统。
- 应用层双活，数据库单活：两个 AZ 的应用程序同时对外提供服务，但是只有一个 AZ 的数据库提供读写，另外一个 AZ 的应用程序需要跨 AZ 访问数据库，数据库之间单向复制。该模式在网络延迟相对低的同城环境下表现良好，但是如果距离超过 100 公里，AZ 之间的网络延迟就会超过 2ms，此时对于跨 AZ 访问的数据库请求，性能有较大影响。

针对同城网络延迟低，可以看作同一个局域网的特例，应用层采用双活，数据库采用单活是相对合理的方案，如图 5-43 所示，应用跨 AZ 访问数据库，一旦某个 AZ 故障，将另外一个 AZ 的应用访问请求切换到本 AZ 的数据库，从而实现同城任何一个数据中心出

现故障，都不影响整体业务的运行。由于同城之间网络条件相对较好，MySQL 数据库原生的复制性能一般能够满足要求，MySQL 5.7 版本推出的并行复制可以有效解决容灾机房日志回放慢的问题。

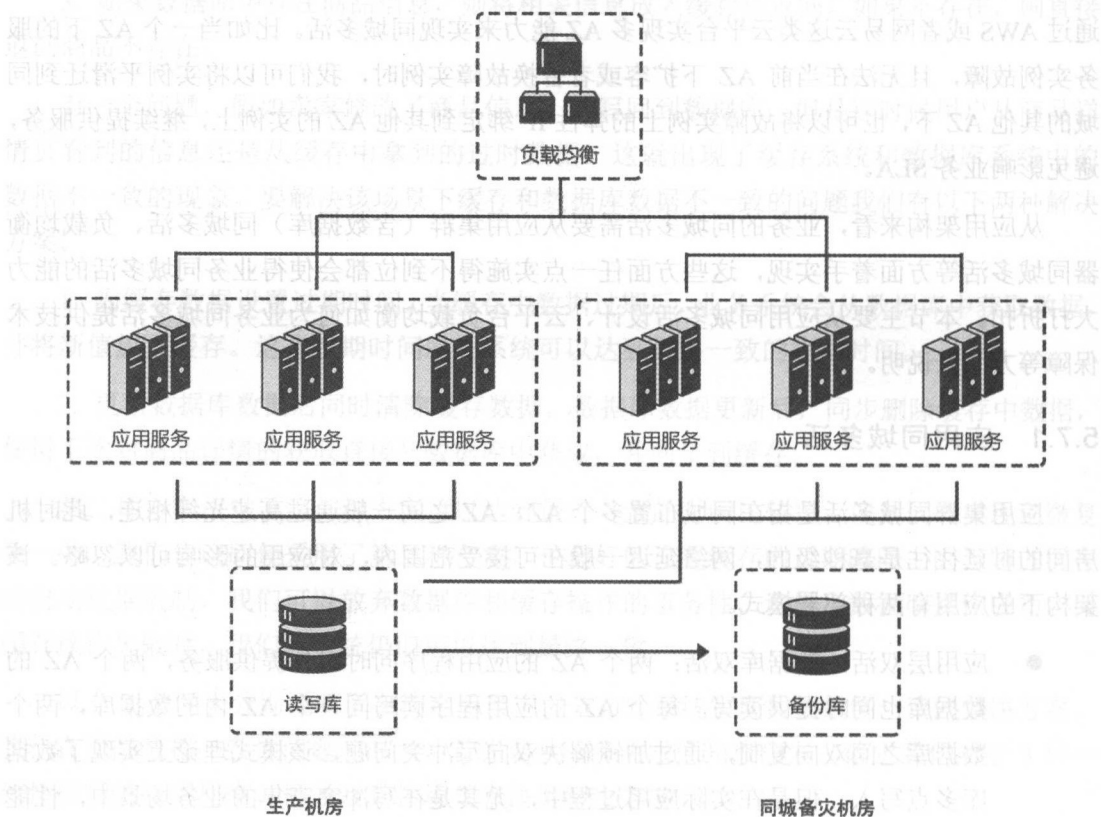


图 5-43 应用层双活、数据库单活方案

同城多活相对异地多活在技术复杂性上已经降低了很多，对业务的侵入较少，适用于跨机房容灾的初级阶段的用户。为了确保方案有效，需要定期进行演练。同时遵循以下原则可以减少实施代价。

1. 只实施核心业务的同城多活。甄别出系统中的主要业务，比如用户中心服务，主要业务有注册和登录，对于一个日活百万的系统来说，很明显登录才是核心业务，因此可以优先实现登录的同城多活。

2. 保证最终一致性。我们通过减少同城机房间距离、减少数据传输量等方式尽量提升

数据同步的实时性，但是因为有物理网络的限制，分布式系统时延必然存在，因此应该从业务层面接受数据的暂时不一致，通过应用保证数据最终一致。

3. 应用和存储系统相结合，避免只采用存储系统的数据同步机制。存储系统自带的同步功能（比如 MySQL 主从复制、Redis 集群方案等）在使用场景上有一定限制，不一定能满足所有业务，因此我们可以改变思路，在应用层使用二次读、回源读等方式实现数据中心数据短暂的不一致问题。

4. 不追求 100% 高可用。跨数据中心的高可用性无法做到 100%，因此我们只能接受这个结果，通过牺牲部分的用户体验来换取。

综合以上讨论，我们可以一句话总结应用同城多活策略的目标：满足大部分用户需求，优先保证核心业务的同城多活。

5.7.2 跨 AZ 负载均衡

对于跨 AZ 负载均衡来说，是将负载均衡服务提到了 AZ 层面，支持同一个负载均衡入口流量在多个 AZ 之间分配。具体分配策略会根据服务商不同而略有不同，但一般可以保证流量在多个 AZ 之间均衡分配，最多再加上一些自定义策略支持（比如权重等）。当某个 AZ 异常时，负载均衡服务会自动将异常 AZ 实例从转发列表中移除，将对应流量转发到其他 AZ 去，根据服务商和负载均衡配置不同，这个切换周期一般在几秒到 1 分钟不等。

正是有了跨 AZ 的调度能力，以及相关基础服务的支持，才让云原生服务可以以较小的代价实现跨 AZ 高可用。这也是云计算在可靠性、扩展性方面同传统运维决定性的差距之一。要知道，所谓的跨 AZ，两地三中心，在传统行业中，只有钱多的金融行业才有机会配备。而借着云计算的东风，同城多活这种高大上的服务也吹入了寻常百姓家。

跨 AZ 负载均衡一般都是通过在区域级别部署流量分发设备，实现流量的跨 AZ 分发。具体到不同云计算厂商，一般会有不同的落地方式，但对于应用层面来说，看到效果是基本相同的。即对于一个负载均衡入口，直接流量导入多个 AZ 中去。

实现要点

相对于传统的跨 AZ 实现，在云计算环境下实现跨 AZ，只要申请对应的负载均衡服务（一般价格会比非跨 AZ 稍贵），再将服务挂载在对应负载均衡后面就可以了。但是在具体实施上，还有以下几点要注意的地方。

- 应用支持：跨 AZ 负载均衡只是解决了多 AZ 流量入口的问题。跨 AZ 还有其他问题需要解决，比如跨 AZ 的通信、数据同步、计算资源分配等问题。
- 容量规划：要做到跨 AZ 容灾，需要保证每个 AZ 的容量余量，当一个 AZ 异常时，其他 AZ 的服务有充足的计算能力可以接手从异常 AZ 转移过来的流量。在单个 AZ 宕机时，不会因为突然的流量导入而导致服务异常。
- 避免 AZ 单点：在实际部署时，需要保证每个 AZ 都可以保证一个系统核心功能正常运行所需要的所有资源，避免多 AZ 环境下的资源单点。需要保证所有的核心资源都是跨 AZ 高可用，当异常发生时可以正常服务。

实践建议

要使用跨 AZ 负载均衡，第一步就是要挑选一家支持多 AZ 负载均衡的提供商。现在较大的云计算提供商基本都提供了跨 AZ 负载均衡的支持。以网易云基础服务为例，提供了 AZ 级别的负载均衡服务，支持单个 VIP 流量在多 AZ 之间转发。而除了传统的主机级别的跨 AZ 转发，网易云基础服务也对容器云跨 AZ 高可用提供了良好的支持，提供了跨 AZ 的服务高可用。

除了选择区域和 AZ 之外，创建跨 AZ 负载均衡与传统负载均衡不会有太多不同。要点在于之后的服务分配和有效性验证。这一点说起来简单，但实际部署的要求很高，需要对整个系统通盘考虑和掌握所有部署细节。

在条件允许的情况下，建议定期进行多 AZ 故障演练和灾备（最好在线上灰度进行），这样才能保证多 AZ 机制可以正常工作。

5.8 故障诊断

在简单的部署环境下，如果线上出现问题，我们一般都需要查看日志，找到跟问题相关的线索，然后根据日志做一些判断，如果不能定位问题，我们可能需要去看操作系统数据或者应用进程相关的数据，比如 JVM 的相关统计数据。但在复杂的部署环境下，如微服务或者分布式服务架构下，结构复杂，服务之间的依赖关系庞杂，涉及的机器或者应用进程很多，一个请求从进入系统到最终返回，可能会经过很多个服务。在这种场景下，我们怎么办？如果每台机器分别查看，效率很低。所以就有了一些日志收集的系统（如 ELK）

把机器的日志都收集起来，通过统一的接口向外提供查询。这种方式只解决了日志分散的问题，没办法把日志跟具体的请求关联起来，当然你也可以在打日志的时候，通过统一的日志规范，在日志中关联类似请求标识符的信息，把一个请求的所有日志关联起来。这样做一方面需要开发介入，另一方面还不够，日志只是排查问题的一个参考因素，机器数据、进程数据、具体某个请求的异常堆栈，以及函数调用性能等都很重要。

全链路跟踪系统就是为了解决这些问题而产生的解决方案。它会把一个请求从进入系统到返回过程中的相关数据，包含函数调用性能、错误日志、异常堆栈、访问缓存、数据库的性能等数据，以及跟应用进程和机器相关的数据都关联起来，形成一个诊断的闭环，微服务场景下，排查问题就变得非常简单。

全链路跟踪系统

全链路跟踪系统会跟踪进入系统的每一个请求，在请求进入系统的时候，分配一个唯一的 ID，并在后续调用其他服务时，一直透传该 ID，保证这个请求在整个系统的处理过程中，通过该 ID 可以把所有相关的数据都串联起来。在全链路跟踪系统中诊断将会变得很简单，你只需要找到某个请求分配的 ID，然后就可以通过这个 ID 找到所有跟这个请求相关的数据，如错误日志、异常堆栈、调用函数性能等。

全链路跟踪系统可以提供一个请求在服务间调用的每一环的性能统计数据。在没有全链路跟踪系统之前，我们没办法统计服务之间的性能，而有了全链路跟踪系统，我们就可以统计服务之间调用的负载、响应时间、错误率等数据。

产品拓扑图

在产品发展的初期，部署结构简单，一般开发人员会通过文档的形式来记录产品的部署结构，但在微服务场景下，服务拆分后会快速增长，一段时间后，就很难再去通过人工的方式来维护这个拓扑结构（开发与运维的脱节，或者开发没有及时跟进等）。而这个拓扑图，对于后续产品的维护及问题的排查很重要。

如图 5-44 所示，在全链路跟踪系统中，我们可以基于服务间的调用数据，自动地发现拓扑图，并能够把拓扑图中每个服务的性能数据关联起来，使产品能够从全局去把控产品的运行状态。

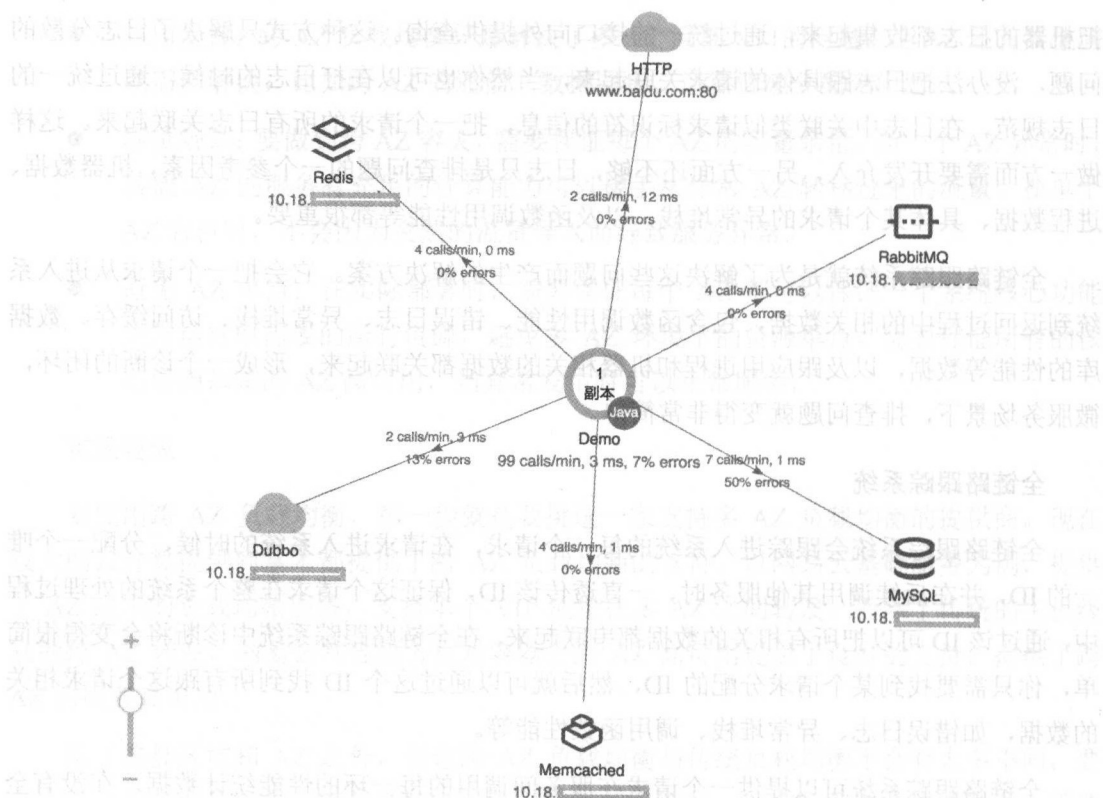


图 5-44 全链路跟踪系统

健康状态

一般情况下，判断应用或者服务整体是否正常，都可以直接通过向用户暴露的应用指标来判断，但很多时候，机器数据或者进程数据也是重要的参考依据。在有全链路跟踪系统支持的情况下，这个判断指标可以更丰富，如服务间的性能是否正常、访问中间件的情况是否正常、服务间的错误率是否异常等，如图 5-45 所示。我们可以根据这些数据对单个服务或者产品整体定义一个整体的健康状态，在系统发生任何异常时，如进程瘫痪、响应变慢、机器运行终止等影响产品整体服务质量的事件时，产生一个健康事件，并及时反馈给产品，帮助用户及时发现问题。

| 异常状态 ↓ | 违背的规则 ↓ | 错误描述 ↓ | 开始时间 ↓ | 结束时间 ↓ |
|--------|--------------------------------|--|---------------------|---------------------|
| ⚠ 一般 | DefaultHealthRule-ResponseT... | /napm-comb-server-test/mockpathtest请求异常: ... | 2016-12-27 09:52:01 | 2016-12-27 11:58:01 |
| ⚠ 一般 | DefaultHealthRule-ResponseT... | /napm-comb-server-test/mockpathtest请求异常: 吞吐量=51.38(大于50.00);平均响应时间=146.79(大于2倍基线标准差) | 2016-12-27 05:29:01 | 2016-12-27 08:47:01 |
| ⚠ 一般 | DefaultHealthRule-ResponseT... | /napm-comb-server-test/mockpathtest请求异常: ... | 2016-12-26 19:35:02 | 2016-12-27 05:28:01 |
| ⚠ 一般 | DefaultHealthRule-ResponseT... | /napm-comb-server-test/mockpathtest请求异常: ... | 2016-12-26 19:05:01 | 2016-12-26 19:34:02 |
| ⚠ 一般 | DefaultHealthRule-ResponseT... | /napm-comb-server-test/mockpathtest请求异常: ... | 2016-12-26 16:03:01 | 2016-12-26 19:04:01 |
| ⚠ 一般 | DefaultHealthRule-ResponseT... | /napm-comb-server-test/mockpathtest请求异常: ... | 2016-12-26 14:45:01 | 2016-12-27 08:47:01 |

图 5-45 健康状态判断指标

慢响应

如图 5-46 所示,传统监控系统中,监控指标主要以绝对阈值来体现,如响应时间是否大于 10ms,但这种指标有一个问题是,不同的应用接口可能完成不同类型的任务,导致这个响应时间差距很大,所以使用基线(通过过去一周每天同一时间的响应时间计算)来判断接口是否正常比较合理。

| 时间 ↓ | 慢响应级别 ↓ | 请求名称 ↓ | 具体的URL ↓ | 执行时间 ↓ |
|-----------------------|---------|--------------------------------|---|--------|
| 📷 2016-12-27 11:42:26 | ⚠ 极慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 352ms |
| 📷 2016-12-27 11:42:24 | ⚠ 极慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 309ms |
| 📷 2016-12-27 11:42:23 | ⚠ 极慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 402ms |
| 📷 2016-12-27 11:42:21 | ⚠ 极慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 308ms |
| 📷 2016-12-27 11:42:20 | ⚠ 极慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 347ms |
| 📷 2016-12-27 11:42:19 | ⚠ 慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 42ms |
| 📷 2016-12-27 11:42:18 | ⚠ 极慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 373ms |
| 📷 2016-12-27 11:42:17 | ⚠ 极慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 344ms |
| 📷 2016-12-27 11:42:17 | ⚠ 极慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 43ms |
| 📷 2016-12-27 11:42:16 | ⚠ 慢 | /napm-comb-server-test/mock... | http://59.111.109.18/napm-comb-server-test/moc... | 39ms |

图 5-46 慢响应监控

请求调用链（请求快照）

通过监控指标异常的方式发现问题，再配合适当的在线诊断工具，可以帮助我们解决线上故障。但是，如果问题偶现，并且很难复现，那么在问题发生时，保留请求的调用信息或者说请求快照，就可以很方便地根据请求的快照信息快速定位问题，而不需要经过复杂的猜测验证来复现问题。

请求快照中包含任何跟这个请求相关的上下文信息，性能访问数据（如函数调用性能、服务间调用性能、访问中间件的性能等），异常堆栈信息，错误日志信息等。图 5-47 就是一个简单的请求快照。

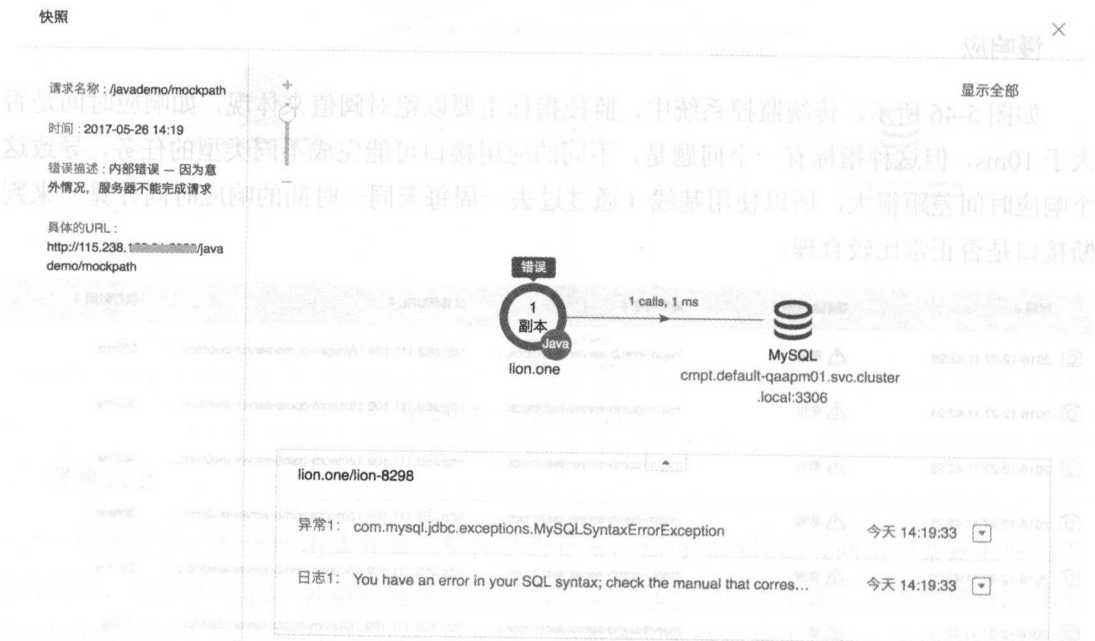


图 5-47 请求快照示例

性能统计

如图 5-48 所示，提供产品整体（对外暴露给用户的接口）或者产品内部各个服务之间的性能调用数据，如吞吐量、响应时间、错误率、错误次数等，通过这些数据，可以比较直观地观察服务质量。

| 请求名称 | 请求次数 ↓ | 请求频率 ↓ | 错误次数 ↓ | 错误频率 ↓ | 平均响应时间 ↓ | 慢响应 ↓ | 极慢响应 ↓ | 无响应 ↓ |
|-------------------------|--------|---------|--------|--------|----------|-------|--------|-------|
| /napm-comb-server-t... | 230644 | 160/min | 28 | 0/min | 82ms | 3916 | 47855 | 100 |
| /napm-comb-server-t... | 2 | 0/min | 0 | 0/min | 5ms | 0 | 0 | 0 |
| / | 11 | 0/min | 11 | 0/min | 0ms | 0 | 0 | 0 |
| /echo.php | 1 | 0/min | 1 | 0/min | 0ms | 0 | 0 | 0 |
| /manager/html | 3 | 0/min | 3 | 0/min | 0ms | 0 | 0 | 0 |
| /myadmin/scripts/set... | 1 | 0/min | 1 | 0/min | 0ms | 0 | 0 | 0 |

图 5-48 性能统计示例

操作系统及应用进程数据

在通过全链路跟踪系统排查问题时，虽然请求快照提供了请求相关的上下文信息及性能数据，但操作系统及应用进程的数据对于问题诊断还是一个比较重要的参考。把单个请求的快照数据跟操作系统及应用进程的数据关联起来，用户就可以很方便地查看这些数据，而不需要每台机器分别去查看，提升效率。

小 结

至此，我们介绍了成熟稳定期业务和架构的几个重要方面。

首先，通过服务拆分的方式避免业务复杂性上升带来的产品交付效率下降，通过统一配置中心解决配置项混乱问题，通过分布式定时任务系统和分布式锁系统解决定时任务和锁系统的单点缺陷，满足大型互联网场景下的相关需求。

其次，总结了微服务架构的实践方式，介绍服务发现、服务治理、服务编排、微服务测试等在微服务架构中如何实践，介绍 Spring Cloud 和 Dubbo 这两个较为流行的微服务框架及它们的应用场景。同时，也对微服务化后出现的数据一致性问题 and 解决方案进行了探讨。

最后，介绍了同城多活这种更高层次的高可用实现方式，以及作为微服务化实践中不可或缺一环的全链路故障诊断和性能分析。

参考文献

- [1] 控制反转. https://en.wikipedia.org/wiki/Inversion_of_control
- [2] MVC 模式. <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>
- [3] 高可用. https://en.wikipedia.org/wiki/High_availability
- [4] Pod. <https://kubernetes.io/docs/concepts/workloads/pods/pod/>
- [5] Replication Controller. <https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/>
- [6] 常见安全风险. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project
- [7] IETF.RFC 2616, <https://tools.ietf.org/html/rfc2616>
- [8] 佚名.Kubernetes Services, <https://kubernetes.io/docs/user-guide/services/>
- [9] 网易云基础服务. 负载均衡, <https://c.163.com/product/balance>
- [10] 网易云基础服务. 缓存服务, <https://c.163.com/product/redis>
- [11] 陈红梅. 互联网信贷风险与大数据架构 (第三版). 北京: 清华大学出版社, 2015.
- [12] 佚名. Docker, wiki, [https://en.wikipedia.org/wiki/Docker_\(software\)](https://en.wikipedia.org/wiki/Docker_(software))
- [13] 佚名. Load balancing, wiki, [https://en.wikipedia.org/wiki/Load_balancing_\(computing\)](https://en.wikipedia.org/wiki/Load_balancing_(computing))
- [14] 佚名, 一篇文章带你了解 Cloud Native, <http://www.open-open.com/lib/view/open1447420363069.html>
- [15] Kubernetes. Horizontal Pod Autoscaling, <https://github.com/kubernetes/community/blob/master/contributors/design-proposals/horizontal-pod-autoscaler.md>
- [16] Sam Newman. 微服务设计. 人民邮电出版社.
- [17] Consul HTTP API. <https://www.consul.io/api/catalog.html>
- [18] Dubbo. <http://dubbo.io/User+Guide-zh.htm>

- [19] Docker. <https://docker.com/>
- [20] Docker Hub. <https://hub.docker.com/>
- [21] Docker — 从入门到实践. https://www.gitbook.com/book/yeasy/docker_practice/details
- [22] 二阶段提交. Wiki. <https://zh.wikipedia.org/wiki/二阶段提交>
- [23] Dan Pritchett. 2008. BASE: An Acid Alternative. Queue 6, 3 (May 2008), 48-55. DOI: <http://dx.doi.org/10.1145/1394127.1394128>
- [24] antirez. Distributed locks with Redis, <http://redis.io/topics/distlock>
- [25] <https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration>
- [26] <http://microservices.io/>

技术术语

PV (Page View): 页面浏览量

UV (User View): 独立访客

DAU (Daily Active User): 日活跃用户数量

MAU (Monthly Active Users): 月活跃用户数量

B2C: 企业对消费者

B2B: 企业对企业

回头率 = 回头客人/店铺总访客数

转换率 = 客户数/UV

点击率 = 点击数/页面展现

PV 利用率 = 客户数/PV

客户月增长率 = 当月客户数/上月客户数 - 1

客单价 = 支付宝成交金额/支付宝成交笔数

页面打开率 = 从首页点击进入宝贝页的次数/首页浏览量

人均访问页面数 = PV/UV (浏览人数/页面总访问人数)

DDoS: 分布式拒绝服务攻击

可靠性: 系统在特定时间特定条件下无故障实现指定功能的能力

康威定律: 组织形式等同系统设计

单体架构: 功能的综合体, 把所有的业务逻辑放在一套代码实现

微服务架构: 一种松耦合的能够被独立开发和部署的无状态化服务 (独立扩展、升级和可替换)

网易云基础服务架构团队，负责网易云基础服务平台建设，包括计算、网络、存储、CDN、数据库等服务，以及自动化平台的架构和实践，该平台支撑了网易内部95%的互联网产品。团队在互联网产品开发和规范化的系统建设上有丰富的经验，对互联网业务需求、系统设计、开发、测试、运维和调优等方面有独到的经验和理解。

专家力荐

从十余年前的各种分布式系统研发到现在的容器云，从支撑原有业务到孵化各个新业务，网易的发展离不开统一的、与时俱进的技术架构。网易云厚积薄发，对外开放内部的技术积累和知识经验，本书是网易架构演进经验的集大成者，希望能够帮助整个产业更好地利用云计算技术推动业务更上一层楼。

——汪源，网易杭州研究院执行院长

处在互联网+浪潮中的技术从业者，需要理解基础架构的前沿，把宝贵的时间和精力集中在业务逻辑和交互实现上。新时代互联网的服务架构已经衍生出全面的技术生态，不仅改变了基础服务的利用方式，也在深刻地改变着应用的开发流程和模式。本书是网易十多年来在多个成功产品上的经验总结，是难得一见的技术分享。愿大家都能从中有所受益。

——翁恺，浙江大学计算机教授

网易博客的流行，网易新闻客户端、网易云音乐、网易云课堂从同类产品中脱颖而出，以及网易考拉海购、网易严选的逐渐崛起，都离不开网易杭州研究院的基础技术与架构，尤其是网易云的支撑。本书作者都是战斗在网易云平台开发一线的主力，这是他们最近几年“云原生应用”开发的最佳总结，相信能让读者收获良多。

——谢骋超，网易考拉海购首席架构师

本书系统讲解在初创期、成长期、稳定期等不同时间，互联网应用在云上的最佳架构实践，技术人一定能找到自己感兴趣的章节，推荐！

——沈剑，58同城技术委员会执行主席

本书基于价值、原则、实践、工具等层次，立体化地诠释云原生，在讨论“12要素应用”、“DevOps”、“持续集成”、“持续交付”等原则、实践层面内容的同时，还介绍了针对不同设计目标场景的工具对比选型和使用方法。本书是网易云基础服务架构团队对多年云计算实践经验的完整总结，也是当前云计算行业对云原生态理念和实践进行深入探讨的有益尝试。

——丁轶群，浙大SEL实验室创始人，杭州谐云科技CTO

网易云架构团队倾其所有、毫无保留地分享他们基于开放技术打造云原生应用架构的经验和心得，阅读此书就像探寻网易内部云原生架构一样，相信读者必受益匪浅。

——王庆，OpenStack基金会个人独立董事，英特尔开源技术中心开发经理

DevOps以业务价值为导向，以持续交付为核心工程实践，力求流水线式、又快又稳地交付有用的价值。而云原生架构使得基础设施等真正变成透明式的“背景”，成为持续交付的基石。网易云团队汇聚多年功力，在本书中阐述了从零到千万并发规模云原生应用的方方面面，您值得拥有。

——萧田国，高效运维社区发起人，DevOpsDays中国联合发起人



博文视点Broadview



新浪微博
weibo.com

@博文视点Broadview



策划编辑：符隆美

责任编辑：徐津平

封面设计：李玲 李倩倩

上架建议：云计算

ISBN 978-7-121-31516-9



9 787121 315169 >

定价：79.00元